Contributions à l'Arithmétique des Ordinateurs : Vers une Maîtrise de la Précision

Marc Daumas

N° d'ordre : 12

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir

le Titre de Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique
au titre de la formation doctorale d'Informatique de Lyon

par **Marc DAUMAS**

CONTRIBUTIONS À L'ARITHMÉTIQUE DES ORDINATEURS
VERS UNE MAÎTRISE DE LA PRÉCISION

Soutenue le 12 Janvier 1996

Après avis de : Monsieur Daniel ETIEMBLE
Monsieur David W. MATULA
Monsieur Jean VIGNES

Devant la commission d'examen formée de :

Monsieur Michel COSNARD
Monsieur Daniel ETIEMBLE
Monsieur Peter KORNERUP
Monsieur David W. MATULA
Monsieur Jean-Michel MULLER
Monsieur Jean VIGNES

Contributions à l'Arithmétique des Ordinateurs :

Vers une Maîtrise de la Précision

Résumé

Depuis l'apparition des premiers ordinateurs, l'arithmétique flottante a énormément évolué. La norme IEEE 754 a permis de fixer les caractéristiques de l'arithmétique des ordinateurs modernes, mais les scientifiques perdent de plus en plus vite le contrôle de la validité de leurs calculs. Malgré l'énorme travail associé à la définition des opérations, la validation des calculs ne peut toujours pas être assurée de façon certaine par l'arithmétique implantée sur les ordinateurs. Je présente dans la première partie de cette étude deux prolongements qui visent à augmenter la marge de validité des opérations : un nouveau mode d'arrondi pour les fonctions trigonométriques et un codage efficace des intervalles accessible facilement à l'utilisateur. Je présente aussi dans cette partie une étude détaillée de la fonction *unit in the last place* et la probabilité d'absorption ou de propagation des erreurs dans une chaîne de multiplication. Ces travaux, qui viennent s'ajouter aux travaux antérieurs d'autres équipes de recherche et aux solutions que j'ai proposées dans ma thèse de *master* montrent les bénéfices que l'on pourra tirer des deux extensions présentées.

L'arithmétique en-ligne permet de gérer efficacement les problèmes de précision, mais les opérateurs élémentaires utilisés sont peu adaptés aux architectures modernes de 32 ou 64 bits. L'implantation efficace d'un opérateur en-ligne ne peut que passer par la description d'un circuit de bas niveau. Les prédiffusés actifs, terme français utilisé pour *Field Programmable Gate Array*, sont des composants spéciaux programmables au niveau des portes logiques. Ils permettent d'abaisser les coûts de production en évitant de fabriquer un prototype. Nous avons implanté grâce à ces technologies les opérateurs simples de calcul en-ligne : addition, normalisation, *etc...* Le Noyau Arithmétique de Calcul En-Ligne (Nacel) décrit dans ce mémoire permet d'implanter les opérations arithmétiques usuelles telles que la multiplication, la division, l'extraction de racine carrée et les fonctions élémentaires trigonométriques et hyperboliques par une approximation polynômiale.

Les architectures à flots de données sont insensibles aux difficultés sur lesquelles butent les concepteurs des ordinateurs modernes : temps d'accès à la mémoire, latence de communication, occupation partielle du pipeline d'instructions. Je décris dans ce document le mode de fonctionnement d'une machine virtuelle appelée Petite Unité de Calcul En-ligne (Puce). Par une gestion adaptée des étiquettes inspirée pour le contrôle des données de celle utilisée par la *Manchester Data Flow Machine*, Puce reproduit le comportement complet d'une machine à flot de données. Elle comprend de plus les opérations en-ligne de calcul scientifique. Nous présentons afin de valider le modèle d'évaluation de Puce les résultats de simulations logicielles pour une ou plusieurs unités fonctionnelles.

Mots-Clés Arithmétique des Ordinateurs, Nombres à Virgule Flottante, Contrôle de la Précision, FPGA, Architecture des Ordinateurs, Machines à Flot de Données

Contributions to Computer Arithmetic :

Towards a Control of Precision

Abstract

Since the apparition of the first computer, floating point arithmetic have drastically changed. The IEEE standard of implementation has been successful in settling the specifications of modern computer arithmetic, but the scientists are loosing rapidly the control and the validation of their computations. In spite of the huge amount of work associated to the definition of the standard operations, the numerical validation of a program cannot always be left to the arithmetic implemented on the computer. I am presenting in the first part of these studies two extensions to offer to the user a higher chance of validity: a new rounding mode adapted to the elementary functions and an efficient coding for the intervals that could be used easily by end users. I present in the first part of my work a detailled definition of the unit in the last place function and the probability of absorption or propagation of round-off errors in a stream of multiplications. This work, added to the former work of many research groups and the proposal contained in my master thesis clearly show the benefits that could be drawn from the proposed extensions.

It is possible with on-line arithmetic to adapt at each step the precision of computation, but the basic operators are not adapted to modern 32 bit or 64 bit architectures. An efficient implementation of on-line operators will necessarily involve a description of a low level circuit. The Field Programmable Gate Array, are electronic component in which each logic part can be configured. By using FPGAs, we are able to lower the cost of production of a new circuit since we do not need a prototype. We have implemented with these technologies the common on-line operators: addition, normalization and so on... The kernel for on-line arithmetic (in French, Nacel) defined in this dissertation has been used to implement more evolved arithmetic operators as the multiplication, the division, the extraction of square root and the elementary functions through a polynomial approximation.

Data-flow architectures are not sensitive to the difficulties that arise in the design of modern computers: memory access, communication latency, partial use of the instruction pipeline. I shall describe in this work the functionnal model for a small on-line arithmetic virtual unit (in French, Puce). By implementing a token mechanism equivalent to the process implemented on the Manchester Data-Flow Machine, Puce is able to reproduce the behavior of a small data-flow machine. We have added to the model the basics for on-line arithmetic, and Puce is able to evaluate on-line any numerical expression. We shall present towards the end of this work the result of a software simulation of Puce on a Fast Fourier Transform to validate the model of evaluation.

Keywords Computer Arithmetic, Floating Point Numbers, Precision Control, Field Programmable Gate Arrays, Computer Architecture, Data Flow Machines

Introduction

Les sciences reposent souvent sur les capacités des chercheurs à établir des modèles mathématiques aptes à représenter au moins partiellement le monde réel. Le calcul numérique est un des moyens d'exploitation des modèles qui permettent de prévoir le comportement de différents systèmes. Cependant, un des obstacles majeurs à nos capacités d'analyse réside dans le manque de certitude sur la fiabilité des résultats : les mesures physiques et les calculs sont entachés d'erreurs qu'il faut maîtriser au mieux.

Très tôt, le monde scientifique a appris à travailler avec ces erreurs. Par exemple, les calculs d'acidité des solutions aqueuses peuvent apporter des résultats de façon satisfaisante en ne prenant en compte que les ordres de grandeur des différents résultats intermédiaires. C'est à la suite de cette habitude qui consiste à gérer les grandeurs numériques en utilisant les premiers chiffres significatifs que l'arithmétique en virgule flottante ou plus simplement l'arithmétique flottante a été conçue.

Depuis l'apparition des premiers ordinateurs, l'arithmétique flottante a énormément évolué, mais les scientifiques perdent de plus en plus vite le contrôle de la validité de leurs calculs. Une quantité énorme d'information peut être traitée par une simple machine et, même un chercheur chevronné, ne peut suivre en détail l'exécution d'un programme. Toutes les opérations de validation et de vérification des résultats intermédiaires qui étaient effectuées autrefois au cours même des calculs ne sont plus possibles et nous nous reposons maintenant presque entièrement sur les opérations arithmétiques disponibles sur nos ordinateurs. Des techniques de validation sont souvent disponibles, mais chaque technique n'est adaptée qu'à quelques types de problèmes. Par ailleurs, ces techniques sont parfois longues à mettre en œuvre et complexes à inclure dans les calculs qu'elles doivent valider.

La norme IEEE d'implantation de l'arithmétique flottante est reconnue à la fois des scientifiques qui ont participé activement aux comités IEEE/ANSI et des industriels qui ont sollicité ce travail. Elle a permis de fixer les caractéristiques de l'arithmétique des ordinateurs modernes. Un très grand soin a été apporté à l'ensemble des opérations requises pour obtenir la conformité à la norme. Chaque opération machine est définie à partir de l'opération mathématique associée et du mode d'arrondi actif. L'ensemble est soutenu par le mécanisme de gestion des arrondis (quatre modes), qui est inclus dans la norme. Les travaux de normalisation ont porté sur deux axes supplémentaires : la construction d'une hiérarchie de types, *single*, *double* et types étendus, et la spécification complète des mécanismes de clôture et d'interruption.

Simultanément au développement de la norme, la puissance de calcul toujours croissante des ordinateurs modernes a fait perdre aux utilisateurs le contact avec leurs calculs. Malgré l'énorme travail associé à la définition des opérations, la validation des calculs ne peut toujours pas être assurée de façon certaine par l'arithmétique implantée sur les ordinateurs. J'ai proposé au cours de mes travaux plusieurs prolongements visant à augmenter la marge de validité des opérations : intégrer les fonctions trigonométriques avec la philosophie qui a présidé à la définition de la norme ; traiter les opérations vectorielles comme des opérations atomiques ; gérer la propagation des erreurs d'arrondi au travers d'un calcul d'intervalles accessible facilement à l'utilisateur *etc*...

Il existe par ailleurs des arithmétiques originales différentes de la virgule flottante qui permettent de gérer efficacement les problèmes de précision. L'arithmétique en-ligne offre bien des possibilités, mais les algorithmes utilisés sont inhabituels et fort sensibles au domaine de définition des arguments. Les recherches entreprises depuis la création de l'équipe Silico-Algorithmes et Arithmétique des Ordinateurs au Laboratoire de l'Informatique du Parallélisme ont permis de clarifier énormément les problèmes théoriques liés à la définition des opérateurs et à l'utilisation de l'arithmétique en-ligne.

Les opérateurs élémentaires utilisés pour l'arithmétique en-ligne sont peu adaptés aux architectures modernes de 32 ou 64 bits. Il est possible de simuler une unité en-ligne sur les ordinateurs

existants même si cette opération ne s'avère intéressante que pour des problèmes très particuliers. L'implantation efficace d'un opérateur en-ligne ne peut que passer par la description d'un circuit de bas niveau. C'est ainsi que j'ai défini par le passé un circuit VLSI permettant la décomposition d'un angle dans la base discrète associée à l'algorithme CORDIC en mode rotation pour calculer son cosinus ou son sinus..

Les prédiffusés actifs, terme français utilisé pour *Field Programmable Gate Array*, sont des composants spéciaux programmables au niveau des portes logiques. Ils permettent d'abaisser les coûts de production en évitant de fabriquer un prototype. Parce qu'ils sont reconfigurables, la fin de la validation d'un circuit et le début de son exploitation peuvent être menés de pair. Si une erreur devait apparaître après le début de l'utilisation du circuit, on peut aisément en modifier la configuration, opération qui est impossible avec un circuit dédié. De plus, un circuit reconfigurable installé sur une machine peut être utilisé pour plusieurs projets différents. Nous avons acquis au LIP un prototype du coprocesseur *PeRLe* construit par le laboratoire parisien de *Digital Equipment*. Le cœur du circuit est composé d'une matrice de calcul de 16 circuits intégrés de type *Xilinx 3090*. Ce noyau est entouré de 7 puces du même type qui se chargent de gérer les communications avec la machine hôte, une station de travail *DEC 5000*. Nous avons implanté sur cette carte les opérateurs simples de calcul en-ligne : addition, normalisation, *etc*... Le Noyau Arithmétique de Calcul En-Ligne (NACEL) décrit dans ce mémoire permet d'implanter les opérations arithmétiques usuelles telles que la multiplication, la division et l'extraction de racine carrée. Nous avons ensuite vérifié que ce noyau peut être utilisé pour évaluer des polynômes par la E-méthode d'Ercegovac et donc calculer efficacement une approximation des fonctions élémentaires trigonométriques et hyperboliques.

Les architectures à flots de données décrites à ce jour sont souvent considérées comme des créations théoriques coupées des applications réelles. Dans les faits ces architectures sont néanmoins insensibles aux difficultés sur lesquelles butent les concepteurs des ordinateurs modernes : temps d'accès à la mémoire, latence de communication, occupation partielle du pipeline d'instructions. Mais grâce aux progrès techniques incéssants réalisés sur les achitectures usuelles, les ingénieurs repoussent toujours les limites du modèle de Von Neumann en adaptant pour certains cas les avancées obtenues à partir des études menées sur les machines à flots de données. Je décris dans ce document le modèle d'une Petite Unité de Calcul En-ligne (PUCE) : par une gestion adaptée des étiquettes, inspirée pour le contrôle des données de celle utilisée par la *Manchester Data Flow Machine*, l'unité virtuelle Puce reproduit le comportement complet d'une machine à flot de données. Le modèle comprend de plus les opérations en-ligne de calcul scientifique. Je présente les résultats de simulations sur des calculs divers pour une ou plusieurs unités fonctionnelles. La génération automatique de code pour Puce par un compilateur n'est pas envisagée ici, mais les travaux effectués sur les langages à affectation unique dans la dernière décennie permettent de prévoir une implantation aisée d'un compilateur, entièrement transparent pour l'utilisateur aux paramètres internes du calcul en-ligne et capable de générer un code efficace pour les microprocesseurs superscalaires ou les machines parallèles.

Le chapitre 1 est consacré aux arithmétiques flottantes usuelles. Il passe en revue quelques problèmes et des extensions possibles. Celles-ci ont été choisies parce qu'elles ne modifient que faiblement le modèle utilisé et sont susceptibles d'être implantées à moyen terme. Une étude détaillée de la carte *PeRLe* est faite dans le chapitre 2. La description complète de la cellule Nacel et l'étude de Puce constituent le corps de ce chapitre qui est consacré à l'arithmétique en-ligne.

Remerciements

Jean Michel Muller a assuré avec efficacité et bienveillance la direction de ma thèse. Cela a été pour moi une chance de travailler avec un chercheur de sa qualité. Jean Michel Muller anime aussi un groupe de travail entre des équipes de Saint-Étienne, Grenoble et Lyon où des domaines très différents de l'arithmétique des ordinateurs sont étudiés. Dès ma deuxième année à l'École, il m'a mis en contact avec d'autres scientifiques de grande valeur qui, comme lui, font progresser l'arithmétique des ordinateurs modernes. Tous m'ont aidé, au cours de ma thèse, de leurs conseils et de leurs compétences techniques. Je voudrais leur adresser mes remerciements.

- J'ai fait mes premières expériences de recherche avec David Matula *deep in the heart of Texas* à SMU (*Southern Methodist University*). La liberté qui m'était donnée, le contact direct avec les problèmes et avec les méthodes de la recherche ont fait de cette année à Dallas une année très fructueuse aussi bien sur le plan professionnel que sur le plan personnel. Tout au long de ma thèse, les conseils et les indications de David Matula ont été précieux pour orienter mon travail.

- Les travaux de Jean Vignes autour des méthodes d'analyse stochastique permettent de mettre en avant facilement les codes numériques qui posent des problèmes. Plusieurs des travaux que je présente ici sont fortements connectés aux questions que Jean Vignes a soulevées ou abordées. Je voudrais le remercier pour l'honneur qu'il m'a fait en acceptant d'être rapporteur de ma thèse.

- Jean Vuillemin m'a encadré pendant trois mois dans le laboratoire parisien de Digital (PRL). Il a fourni, avec la carte que son équipe a conçue, le *medium* d'implantation de mon travail. Ce court passage à Digital, dans un laboratoire peuplé de polytechniciens, a été d'autant plus enrichissant que les membres du groupe de Jean Vuillemin étaient toujours ouverts à mes questions et disposés à m'aider.

- Après que Daniel Etiemble ait lu mon mémoire, nous nous sommes engagés dans un échange très enrichissant. J'ai ainsi pu réorienter certaines points de ma thèse en regard des travaux les plus récents en architecture des ordinateurs. J'espère pouvoir continuer ces travaux dans les directions que nous avons identifié en commun comme très prometteuses.

- Je n'ai jamais eu l'occasion de travailler directement avec Peter Kornerup, ce que je regrette vivement car, à chaque fois qu'il m'a été donné de le rencontrer, il s'est montré à l'écoute de mes problèmes. Les questions que j'ai pu lui adresser au sujet de ses recherches ont toujours reçu un excellent accueil.

Je voudrais adresser tout spécialement mes remerciements à Michel Cosnard, ancien responsable de l'enseignement d'informatique à l'École, fondateur et directeur du LIP qui m'a accueilli pendant mes travaux. Michel Cosnard a été pendant les années où j'ai été présent à l'École, le moteur de l'informatique. J'ai souvent eu à faire appel à lui au cours de ma vie d'étudiant, aussi bien comme élève que dernièrement comme thésard.

Faisant partie des premières promotions de l'École Normale Supérieure de Lyon, j'ai pu constater tout au long de ma scolarité le travail remarquable fait par l'ensemble du personnel de l'administration. Je tiens à souligner le travail quotidien de M-J. Barrier et de J-A. Losdat qui m'ont plus d'une fois aidé dans mes tâches administratives. Je voudrais aussi m'excuser une fois de plus auprès de J. Richerd et V. Roger pour les nombreuses fois où je l'ai dérangée en dehors des heures d'ouverture du secrétariat.

Mon travail de thèse aurait probablement été beaucoup moins fructueux sans l'aide de mes camarades et des enseignants et chercheurs qui m'entouraient. Je voudrais profiter de cette occasion pour les remercier à nouveau de leur aide et de leurs conseils :

- Il existe en France des côtes où il fait beau temps toute l'année (il peut se passer quatre mois sans qu'il ne tombe une goutte d'eau) et où les stations de ski sont à deux heures de route. Et puis il existe, à l'extrémité ouest de notre belle terre une province où il pleut toute l'année, où les plus hauts sommets *culminent* à des altitudes dérisoires, et où la température de l'eau n'atteint même pas les 16 degrés un 1er septembre. J'ai rencontré à Lyon un habitant de cette terre reculée. Jean François pratique fort bien l'escalade, la planche à voile et le ski, ce qui en fait un guide rêvé.

- J'ai utilisé abondamment les premiers résultats de Christophe Mazenc sur la multiplication en ligne sur machine parallèle pour mon travail personnel. Nous avons défriché ensemble de nombreux sujets dont le codage compact des intervalles flottants.

- Je ne voudrais surtout pas oublier les chercheurs et les hôtes de passages de l'équipe SAAO parmi lesquels Anne, Xavier, Jean-Claude, Mike et Asger ni ceux de l'équipe enseignante avec qui j'ai travaillé et dont j'ai pu apprécier l'analyse pédagogique pointue et parfois même acerbe : Odile, Anne, Annick, Jean et les innénarables Luc et Yves.

Je dois avouer à ce stade que je ne suis pas l'auteur de la totalité de ce document. Les illustrations de tête de chapitre sont dues à Stéphane Labbé. Quelques figures ont été inspirées des rapports rédigés par Arnaud Tisserand et Pascal Tock au cours de leurs stages respectifs de DEA et de maîtrise, mais j'ai eu vent que certaines de ces figures avaient déjà été publiées dans des travaux antérieurs.

Chapitre 1

Arithmétique Flottante

Aujourd'hui, la norme d'implantation de l'arithmétique flottante [IEEE 85] est disponible sur la plupart des ordinateurs. Historiquement, la définition de la norme est liée à la construction des ordinateurs personnels dans les années 80. On observe toujours que ces ordinateurs (PC, Mac) offrent une implantation plus soignée de l'arithmétique flottante que les stations de travail, avec le type *double extended* de 80 bits. Certains ordinateurs personnels disposent de plus d'un arrondi affiné des fonctions trigonométriques [FB 91]. Une arithmétique compatible avec la norme est aujourd'hui accessible sur tous les ordinateurs personnels et les stations de travail. Sur quelques super-calculateurs dont des machines Cray, elle n'est encore qu'émulée au niveau logiciel. On peut penser que la prochaine génération d'ordinateurs sera compatible dans son ensemble avec les spécifications de la norme.

La force de cette norme a été d'imposer pour toutes les machines une spécification mathématique claire et concise relativement indépendante des problèmes d'implantation. L'accroissement de la puissance des ordinateurs fait que, à nouveau, les utilisateurs atteignent ou dépassent les possibilités théoriques et pratiques de leurs machines sans en être prévenus. Augmenter la précision des opérations disponibles sur les ordinateurs ne constitue pas une solution à long terme pour ce problème. On constate qu'une énorme majorité des codes utilisés à ce jour n'utilisent plus que l'arithmétique double précision. Sans beaucoup de considération pour les problèmes d'encombrement de la mémoire et de temps d'exécution, les utilisateurs de l'informatique numérique ont naturellement fait évoluer leurs codes du format *single* au format *double*. Cette opération très coûteuse n'est souvent pas validée dans le détail par des données théoriques ou expérimentales. Quand certaines validations ont été réalisées, elles traitent le code dans son ensemble, plutôt que d'en extraire les seules données critiques, ce qui est très difficile avec les machines actuellement disponibles.

L'implantation en machine du format *quad* entraînera le même type de migration. Pour bon nombre de codes, le choix de la précision sera imposé sans qu'il soit possible d'effectuer une étude suffisamment poussée. On assistera alors à une augmentation de la précision du résultat de quelques chiffres. Mais les programmes numériques occuperont tous deux fois plus de place en mémoire et prendront presque deux fois plus de temps pour augmenter la précision de certains résultats intermédiaires qui étaient déjà suffisamment précis en double précision. Pour cette raison, il est essentiel d'étendre la norme dans des directions sans aucun lien avec la taille des mantisses afin de permettre enfin à l'utilisateur de gérer les erreurs générées par les systèmes de calcul [Boh 90]. Il pourra ainsi faire le meilleur usage possible des différents formats offerts et recourir seulement lorsque cela s'avère nécessaire au format *quad* de 128 bits qui restera très coûteux.

La section 1 propose une vue d'ensemble des données nécessaires à une analyse convenable des problèmes de la norme IEEE. La quantité *unit in the last place* (ulp), a déjà été définie par le passé sous diverses appellations [WF 82, Gol 91, DM 93a, DM 93b]. Une approche rigoureuse de cette valeur pose néanmoins des problèmes de modélisation que nous aborderons dans cette section. Nous analysons dans la section 2 des cas d'absorption et de propagation des erreurs d'arrondi dans la multiplication en virgule flottante. L'étude d'un problème suggéré par S. Kla qui présente des fortes disparités d'absorption est proposée dans cette section. Un nouveau mode d'arrondi pour les intervalles de petite taille est présenté dans la section 3. Le dernier sujet abordé dans ce chapitre est une proposition originale de codage des intervalles. Nous détaillons dans la section 4 une méthode de codage qui reprend les idées qui ont fait le succès de la norme. Nous avons appliqué avec C. Mazenc ces idées aux intervalles.

1.1 Norme IEEE 754

1.1.1 Trois Concepts Essentiels

La norme IEEE 754-1985 fixe les conditions d'implantation de l'arithmétique flottante binaire. La norme repose essentiellement sur trois concepts qui ont achevé de démontrer leur importance avec le recul du temps : hiérarchie de types, clôture des opérations, reproductibilité des calculs. La norme 854 rédigée par le même comité reprend ces notions pour les arithmétiques flottantes écrites dans une base de numération 2 ou 10 et autorise de plus grandes libertés [IEEE 87]. L'étude que nous proposons dans ce chapitre est basée sur l'arithmétique en base 2. L'extension de cette étude à une autre base de numération est laissée au lecteur car elle ne pose pas de problème particulier.

Hiérarchie de Types Chaque système d'écriture des nombres possède une classe d'algorithmes auxquels il est bien adapté. Le choix d'un système pour une machine donnée représente un compromis entre l'efficacité du système au niveau le plus bas (taille du circuit, opérations élémentaires) et les facilités d'implantation des fonctions arithmétiques les plus utilisées. Une définition homogène du codage des nombres flottants au niveau du bit permet des échanges aisés entre des machines différentes. La norme définit précisément la signification, la position et la taille de chacun des champs qui composent un nombre flottant (voir Figure 1.1). La seule différence possible entre deux représentations sur des machines différentes est due à l'ordre des octets dans les mots en machine — *big endian* ou *little endian*.

- Le premier champ contient le bit de signe s. Un nombre est décomposé en un couple (signe, magnitude). La magnitude est stockée dans les champs suivants.
- L'exposant e se situe juste après le signe. L'exposant peut avoir n'importe quelle valeur entière dans l'intervalle $[-e_{\max}+1, e_{\max}]$ qui est défini pour chaque type. La valeur de e n'est pas stockée à l'aide d'un entier signé en complément à 2. En machine, on a recours à un exposant biaisé appelé par certains auteurs la caractéristique du nombre [AEGP 67]. À la place de e, l'ordinateur stocke la valeur $e + e_{\max} + 1$. Ainsi, la caractéristique d'un nombre est toujours strictement supérieure à 0. La valeur du biais et la taille du champs de l'exposant détaillés Figure 1.3 sont des données du type.
- La fraction f est stockée en dernier dans le mot. La mantisse du nombre s'écrit à l'aide de la fraction $m = 1.f$. L'unité de calcul flottant travaille uniquement sur des nombres normalisés, c'est à dire $1 \leq m < 2$. Le premier bit de la mantisse est donc toujours égal à 1 ; par souci d'économie, on ne le conserve pas en mémoire pour les types usuels. Il faut alors définir un codage spécial pour le nombre 0. La taille ν de ce champ dépend du type de donnée (voir Figure 1.3). Pour les types étendus, on stocke directement la mantisse pour pouvoir utiliser des nombres dénormalisés qui interviennent dans certains algorithmes [DM 93a].

La valeur d'un nombre flottant s'exprime donc par la formule ci-dessous. Les chiffres de f sont numérotés de gauche à droite, en commençant par f_1. Ainsi, les nombres positifs flottants sont rangés dans l'ordre lexicographique de leur écriture. Nous avons représenté Figure 1.2 tous les nombres positifs qui peuvent être écrits ainsi avec une mantisse de quatre bits et un exposant sans biais entre -1 et 1.

$$x = (-1)^s \times 1.f \times 2^e = (-1)^s \times (1 + \sum_{i=1}^{\nu} f_i 2^{-i}) \times 2^e$$

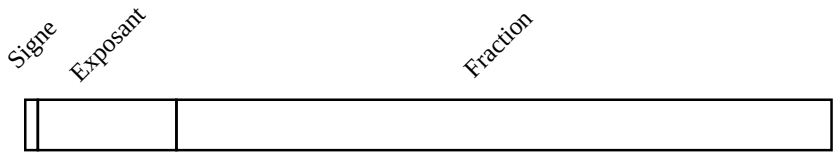


FIG. 1.1 - *Codage Normatif des Nombres (Formats IEEE)*

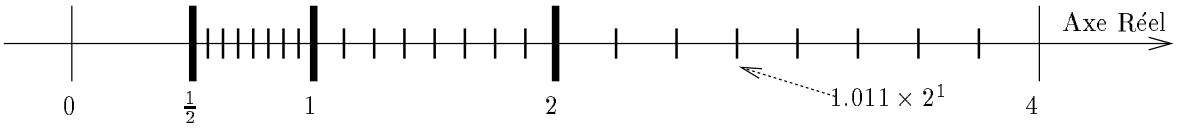


FIG. 1.2 - *Nombres Positifs Normalisés codés sur 3 Bits (Exposant entre -1 et 1)*

Cette façon d'interpréter un nombre flottant x nécessite de manipuler couramment des fonctions relatives à 1 ulp, quantité que nous allons définir et étudier plus en détail dans la section suivante, et de maîtriser les mécanismes d'arrondi à un ulp adjacent. Pour éviter de manipuler de telles quantités, on décide d'appeler *mantisse entière*, M, le nombre entier obtenu par la concaténation :

$$M = 1 \oplus f$$

Les deux représentations basées sur m et M sont équivalentes. Soit l_m la taille en bits du champ réservé pour f plus 1, on obtient la valeur de x en fonction de M par la formule qui suit. On vérifie de plus que f satisfait $0 \leq f \leq 2^{l_m} - 1$ et donc $2^{l_m} \leq 1 \oplus f \leq 2^{l_m+1} - 1$ pour la mantisse entière et $1 \leq 1.f \leq 2 - 2^{-l_m+1}$ pour la mantisse usuelle. Nous utiliserons indifféremment dans ce travail la notation entière proposée ici, que l'on retrouve notament dans [Gos 71], et la notation fractionnaire usuelle.

$$x = (-1)^s \times 1 \oplus f \times 2^{e-l_m+1} = (-1)^s \times (2^{l_m} + \sum_{i=1}^{l_m} f_i 2^{l_m-i}) \times 2^{e-l_m+1}$$

Les structures de données des nombres flottants ne sont pas considérées de façon isolée. Elles sont intégrées à une hiérarchie. C'est ainsi que pour chaque type T (simple, double...) on définit les caractéristiques minimales du type étendu correspondant T^+. Le type T', suivant du type T dans la hiérarchie, devra au moins contenir le type T^+, type étendu de T. Cela permet d'intégrer au niveau logiciel un type *quad* compatible avec la norme comme le font certains constructeurs alors que le type *quad* n'est défini qu'implicitement dans les documents de normalisation. Les types usuels implantés sur les différents ordinateurs sont présentés Figure 1.3. Les unités flottantes des ordinateurs personnels sont capables de travailler avec un type double étendu de 80 bits. Ce type a été mis en place pour permettre d'arrondir avec une sécurité accrue les fonctions élémentaires (sinus, exponentielle...) sur ces architectures et a été repris à ce jour sur toutes les unités flottantes destinées aux ordinateurs personnels.

Type	Mantisse (bits)	Exposant (bits)	Biais	Longueur totale (octets)
Single	$1+23$	8	127	4
Single étendu	≥ 32	≥ 11	≥ 1023	≥ 5
Double	$1+52$	11	1023	8
Double étendu	≥ 64	≥ 15	≥ 16383	≥ 10
Double étendu PC	$1+64$	15	16383	10
Quad	112	15	16383	16

FIG. 1.3 - *Types Flottants Usuels*

Clôture Pour permettre une gestion continue du flot de données, la norme spécifie un résultat pour chaque opération sur l'ensemble du domaine $\mathbb{F}$ des nombres flottants. Les opérations mathématiques renvoient un résultat appartenant à la clôture $\overline{\mathbb{F}}$ du domaine pour les opérations interdites. Des valeurs non numériques ont été ajoutées au domaine pour gérer correctement les opérations qui pourraient générer une erreur. Les valeurs non numériques sont signalées par leur exposant comme nous le voyons Figure 1.4.

+Inf, -Inf $[+\infty, -\infty]$ Dans la norme IEEE, l'ensemble $\overline{\mathbb{F}}$ possède deux valeurs infinies projectives. Ces valeurs représentent à la fois un dépassement de capacité positif ou négatif et les résultats respectifs de $\frac{a}{0^+}$ et $\frac{a}{0^-}$ où $a \in \mathbb{F}_+^*$. L'existence d'un symbole unique pour représenter à la fois une valeur infinie et un dépassement de capacité peut être une cause d'inconsistances dans le cours des calculs [LS 93]. On pourra ainsi évaluer la fonction suivante en simple précision :

$$f(x) = \frac{1+x^2}{\sqrt{1+x^3}}$$

La limite de cette fonction quand x tend vers $+\infty$ est $+\infty$. On constate néanmoins que pour des valeurs comprises entre 10^{13} et 10^{18} la fonction renvoie invariablement 0 comme résultat. En effet, quand la machine calcule x^3, un dépassement de capacité se produit et le résultat est remplacé par `+Inf`. Le dénominateur se réduit lui aussi à `+Inf`. Comme le calcul du numérateur ne provoque pas de dépassement de capacité, l'unité calcule le quotient entre une quantité finie non nulle et un nombre infini. On obtient bien 0. L'unité flottante ne prévient l'utilisateur d'un problème possible que pour des nombres plus grands : le résultat est alors *Not a Number*.

+Zéro, -Zéro $[0^+, 0^-]$ La présence de deux codages différents de zéro est une conséquence directe de la définition des deux valeurs projectives $+\infty$ et $-\infty$ distinctes. La règle des signes s'applique pour ces nombres, ainsi $0^+ = (-1) \times 0^-$. On note néanmoins que le test flottant $0^+ = 0^-$ est vrai, bien que 0^+ et 0^- n'aient pas le même codage en machine.

Not a Number $[NaN]$ Cette valeur sert à coder des résultats non exprimables dans le système défini tels que $\sqrt{-1}$, ou qui ne peuvent être réduits comme $\frac{0}{0}$ ou $\infty - \infty$. Une fois que la valeur NaN apparaît dans le cours d'un calcul tous les calculs qui dépendent de cette variable ont pour résultat NaN. On notera aussi qu'un test dont une des opérandes est NaN est faux sauf pour le cas de l'égalité. Ainsi les test $a < b$ et $a \geq b$ ne sont pas exactement contraires.

Exposant	Fraction	Signification
$e_{max}+1$	0	$\pm\infty$
$e_{max}+1$	Non nulle	*NaN*
$-e_{max}$	Non nulle	Dénormalisé
$-e_{max}$	0	0

FIG. 1.4 - *Valeurs non Numériques*

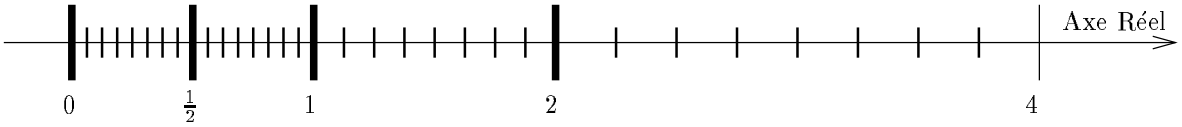


FIG. 1.5 - *Nombres Positifs codés sur 3 Bits (Domaine Étendu)*

Dénormalisé Le nombre 0 est codé en mettant tous les bits du mot machine à 0 sauf éventuellement le bit de signe. L'exposant biaisé de 0 correspond donc à l'exposant $-e_{\max}-1$ soit $-2^{-10}+1$ en double précision. On peut constater Figure 1.2 qu'il y a un saut important entre le dernier nombre représentable non nul et zéro. Les nombres dont l'exposant est $-e_{\max}$ ne sont pas autorisés par construction. On se sert des mots ainsi laissés sans emploi pour diminuer le saut qui a lieu lors d'un dépassement de capacité vers 0. Les nombres d'exposant $-e_{\max}$ codent des valeurs où le 1 implicite au début du mot est absent et où l'exposant est $-e_{\max}+1$. Pour ces raisons, on définit la valeur d'un nombre dont l'exposant biaisé est 0 comme suit :

$$x = (-1)^s \times 0.f \times 2^{-e_{\max}+1} = (-1)^s \times f \times 2^{-e_{\max}-l_m+2}$$

On obtient un spectre beaucoup plus régulier des nombres à l'approche de zéro (voir Figure 1.5). On pourra essayer de calculer la valeur suivante en précision double avec $x \in [10^{-53}, 10^{-52}]$.

$$\sqrt{\frac{1}{\sin^3(x^2)}}$$

Le résultat est alors `+Inf`. Si on lui préfère $\frac{1}{\sqrt{\sin^3(x^2)}}$ qui fait intervenir un nombre dénormalisé, le résultat reste équivalent à $\frac{1}{x^3}$ ce qui est conforme aux calculs même si l'on peut remarquer que la précision du résultat décroît rapidement.

Le traitement des opérations interdites ou erronées se fait par le biais des mécanismes d'exception. Suivant l'état de configuration de l'unité de calcul flottant, une opération illégale pourra déclencher une interruption en machine ou non. Dans le second cas, le résultat affiché à l'utilisateur est donné par une des valeurs non numériques définies par défaut dans la norme (voir Figure 1.6, Figure 1.7 et Figure 1.8). Par ce biais, l'utilisateur est maître du traitement ou de l'absence de traitement des exceptions numériques.

Reproductibilité La spécification totale des mécanismes d'arrondi, incluse dans la norme IEEE a permis de garantir automatiquement la reproductibilité des calculs. Les opérations implantées

Addition	$-\infty$	$v \in \mathbb{F}^*_-$	0^-	0^+	$v \in \mathbb{F}^*_+$	$+\infty$	NaN
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN	NaN
$w \in \mathbb{F}^*_-$	$-\infty$	$v+w$ ou $-\infty$	w	w	$v+w$	$+\infty$	NaN
0^-	$-\infty$	v	0^-	0^+	v	$+\infty$	NaN
0^+	$-\infty$	v	0^+	0^+	v	$+\infty$	NaN
$w \in \mathbb{F}^*_+$	$-\infty$	$v+w$	w	w	$v+w$ ou $+\infty$	$+\infty$	NaN
$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN

FIG. 1.6 - *Addition Flottante*

Multiplication	$-\infty$	$v \in \mathbb{F}^*_-$	0^-	0^+	$v \in \mathbb{F}^*_+$	$+\infty$	NaN
$-\infty$	$+\infty$	$+\infty$	NaN	NaN	$-\infty$	$-\infty$	NaN
$w \in \mathbb{F}^*_-$	$+\infty$	$v \times w$ ou $+\infty$	0^+	0^-	$v \times w$ ou $-\infty$	$-\infty$	NaN
0^-	NaN	0^+	0^+	0^-	0^-	NaN	NaN
0^+	NaN	0^-	0^-	0^+	0^+	NaN	NaN
$w \in \mathbb{F}^*_-$	$-\infty$	$v \times w$ ou $-\infty$	0^-	0^+	$v \times w$ ou $+\infty$	$+\infty$	NaN
$+\infty$	$-\infty$	$-\infty$	NaN	NaN	$+\infty$	$+\infty$	NaN

FIG. 1.7 - *Multiplication Flottante*

Opération unaire	$-\infty$	$v \in \mathbb{F}^*_-$	0^-	0^+	$v \in \mathbb{F}^*_+$	$+\infty$	NaN
Inverse	0^-	$\frac{1}{v}$	$-\infty$	$+\infty$	$\frac{1}{v}$	0^+	NaN
Racine Carrée	NaN	NaN	0^-	0^+	$\sqrt{v}$	$+\infty$	NaN

FIG. 1.8 - *Inverse et Racine Carré Flottante*

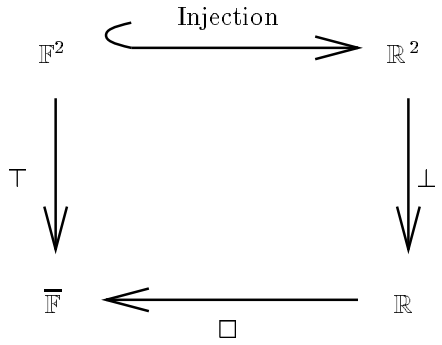


FIG. 1.9 - *Définition d'une Opération Flottante*

ne sont plus définies en fonction des caractéristiques physiques de la machine utilisée mais par des contraintes mathématiques. La précision du résultat dépend alors du modèle mathématique et non des détails de l'implantation. De ce fait, à une précision donnée, un calcul donne un résultat unique pour toutes les exécutions et sur toutes les machines compatibles avec la norme.

Le mode d'arrondi actif $\Box$ est utilisé pour l'opération en cours parmi les quatre modes définis par la norme: arrondi vers 0 — Round towards Zero ($\bowtie$) ; arrondi positif vers $+\infty$ — Round Up towards $+\infty$ ($\triangle$) ; arrondi au plus proche — Round to Nearest ($\circ$) ; arrondi négatif vers $-\infty$ — Round Down towards $-\infty$ (∇). Les opérations flottantes sont définies directement à partir de l'opération mathématique correspondante.

Addition	$\mathbb{F}^2 \longrightarrow \overline{\mathbb{F}}$	$(v,w) \longmapsto \Box(v+w)$
Multiplication	$\mathbb{F}^2 \longrightarrow \overline{\mathbb{F}}$	$(v,w) \longmapsto \Box(v \times w)$
Division	$\mathbb{F}^2 \longrightarrow \overline{\mathbb{F}}$	$(v,w) \longmapsto \Box(v/w)$
Racine Carrée	$\mathbb{F} \longrightarrow \overline{\mathbb{F}}$	$v \longmapsto \Box(\sqrt{v})$

On observe ainsi que toute opération machine $\top$ peut être spécifiée comme l'indique le diagramme commutatif Figure 1.9. Il s'agit de la composition de l'opération mathématique correspondante $\bot$ et du semi-morphisme $\Box$ de $\mathbb{R}$ dans $\overline{\mathbb{F}}$. Pour de plus amples détails sur la notion de semi-morphisne, le lecteur pourra se réferer à [KM 81].

1.1.2 Fonction Ulp

L'ensemble $\mathbb{F}$ des nombres représentables pour un type flottant donné est discret dans $\mathbb{R}$ (voir Figure 1.5). On définit la quantité *least significant bit* obtenue par la différence entre un nombre flottant $v \in \mathbb{F}$ et le nombre v' ayant même codage à l'exception du dernier bit de la mantisse :

$$LSB = |v - v'|$$

L'ensemble $\mathbb{F}$ est un sous ensemble de $\mathbb{R}$. Pour tout réel x, on définit le successeur strict de x et le prédécesseur strict de x dans $\overline{\mathbb{F}}$, respectivement x^+ et x^-.

$$\begin{aligned} x^+ &= \min\{v \in \overline{\mathbb{F}} | v > x\} \\ x^- &= \max\{v \in \overline{\mathbb{F}} | v < x\} \end{aligned}$$

Nous avons représenté Figure 1.10 un nombre flottant $v \in \mathbb{F}$, son successeur strict v^+ et son prédécesseur strict v^- sur l'axe réel $\mathbb{R}$ (vertical). Les traits gras représentent les points de $\mathbb{F}$ — nombres

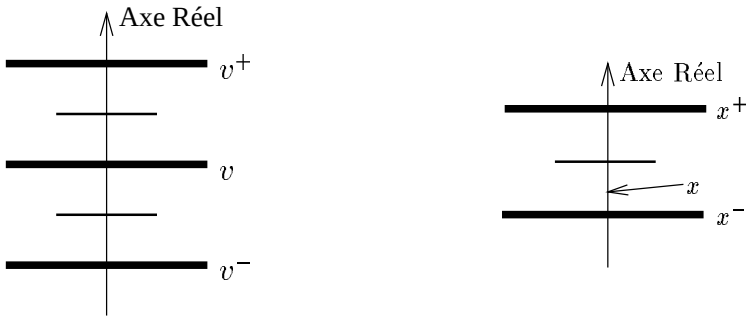


FIG. 1.10 - *Nombres Flottants et Nombres Réels*

codables en virgule flottante, alors que les points intermédiaires à mi-intervalle sont représentés en trait léger. Ces points sont utilisés pour déterminer l'arrondi au plus proche d'un nombre réel. Si $v \in \mathbb{F}$, tout procédé d'arrondi qualifiable doit vérifier $\square(v) = v$. Dans le cas contraire, $x \in \mathbb{R} - \mathbb{F}$, un arrondi convenable devra satisfaire $\square(x) \in \{x^+, x^-\}$.

Nous retenons dans ce travail une autre définition de la grandeur LSB sous la forme de la fonction *unit in the last place*. Pour un nombre flottant $v \in \mathbb{F}$, on définit la fonction $ulp(v)$ à partir des différences $(v^+ - v)$ et $(v - v^-)$. Ces deux quantités ne sont pas toujours identiques et il peut arriver qu'une des valeurs soit infinie. Pour les valeurs flottantes $v \in \mathbb{F}$ telles que $v = \pm 2^k$ avec $k \in \mathbb{Z}$, il y a un facteur 2 entre les deux valeurs, par exemple si $v > 0$ on vérifie que $v^+ - v = 2 \times (v - v^-)$. Pour la valeur flottante $v \in \mathbb{F}$ telle que $v^+ = +\infty$, nous définissons la fonction $ulp(v)$ à l'aide de la propriété de linéarité de la fonction ulp par une homothétie de rapport 2. La symétrie de la fonction ulp nous est imposée par la notation (signe, magnitude) des nombres flottants. Ainsi c'est la fonction paire suivante :

$$ulp(v) = \begin{cases} v^+ - v & \text{si} \quad v \geq 0 \qquad \text{et} \quad v^+ \neq +\infty \\ 2\, ulp\left(\frac{v}{2}\right) & \text{si} \quad v \neq +\infty \quad \text{mais} \quad v^+ = +\infty \\ ulp(-v) & \text{si} \quad v < 0 \end{cases}$$

On peut définir la valeur $ulp(v)$ à partir des champs qui composent l'écriture de v en machine ou à l'aide des fonctions mathématiques usuelles. Dans tout ce travail, en l'absence de risque de confusion, nous noterons log la fonction logarithme à base 2. La valeur $ulp(1)$ apparaît comme une grandeur caractéristique du format déjà utilisée par certains auteurs [CFB 93]. Dans le reste de ce travail, nous noterons également $ulp = ulp(1)$. Soit un nombre flottant $v \in \mathbb{F}$ codé $v = (s, m, e)$, on a :

$$ulp(v) = ulp((s, m, e)) = ulp \times 2^e = (0, 0.0 \ldots 01, e) = ulp \times 2^{\lfloor \log |v| \rfloor}$$

On vérifie que la fonction ulp est constante sur les intervalles de la forme $[2^i, 2^{i+1}[$ et $]-2^{i+1}, -2^i]$ pour tout $i \in \mathbb{Z}$. De plus, on déduit la propriété suivante de la répartition des nombres flottants : soit $v \in \mathbb{F}$ et $k \in \mathbb{Z}$ tel que $2^k \times v \in \mathbb{F}$, alors,

$$ulp(2^k v) = 2^k\, ulp(v)$$

Pour tout nombre réel $x \in \mathbb{R}$, on peut écrire $x = (s, m, e)$ en virgule flottante où m est éventuellement infini et e n'est pas limité par e_{max}. La valeur de LSB de x est donnée comme suit si

l'on code x' par $x' = (s, m', e)$ où m' est obtenu en changeant dans m le dernier bit entrant dans une mantisse finie de taille normalisée.

$$LSB = |x - x'|$$

Pour un nombre réel quelconque $x \in \mathbb{R}$, nous définissons la valeur $ulp(x)$ comme suit par analogie avec la formule définissant ulp sur $\mathbb{F}$. La quantité LSB coïncide avec la fonction ulp pour tous les nombres réels $x \in R$.

$$ulp(x) = ulp(1) \times 2^{\lfloor \log |x| \rfloor} = ulp \times 2^{\lfloor \log |x| \rfloor}$$

Nous avons montré que certains algorithmes nécessitent le recours en dehors de la norme IEEE à des nombres dénormalisés [DM 93a]. On appelle nombre dénormalisé le triplet $v = (s, m, e)$ où $m < 1$. Avec la possibilité de coder des nombres où $v < 1$, le système ne garantit plus une unique écriture pour tous les nombres réels représentés. Par exemple, 2 peut s'écrire 1×2^1 ou 0.5×2^2. On doit alors faire la distinction entre le nombre machine et le nombre réel qu'il représente. Soit k tel que $1 \leq m \times 2^k < 2$, les nombres machines (s, m, e) et $(s, m \times 2^k, e - k)$ ont la même valeur, mais le second est normalisé. On définit $ulp(s, m, e)$ comme suit.

$$ulp(s, m, e) = 2^{-k} \times ulp(s, m \times 2^k, e - k)$$

Pour un nombre flottant $v \in \mathbb{F}$, nous noterons $\square^{-1}(v)$ l'ensemble des nombres réels x qui, une fois arrondis, valent v.

$$\square^{-1}(v) = \{x \in \mathbb{R} | \, \square(x) = v\}$$

La taille de l'intervalle $\square^{-1}(v)$ est $ulp(v)$ sauf dans le cas où $v \in \mathbb{F}$ est une puissance exacte de 2, c'est à dire que $v = \pm 2^k$ avec $k \in \mathbb{Z}$. L'intervalle $\square^{-1}(v)$ représente le maximum de l'erreur d'arrondi d'une opération. Dans le cas où $v = \pm 2^k$, $\square^{-1}(v)$ peut aussi avoir une largeur de $\frac{1}{2}\, ulp(v)$, $\frac{3}{4}\, ulp(v)$ ou $2 \times ulp(v)$. Mais la valeur de $ulp(x)$ ne peut pas être définie en grandeur relative à x, en effet $ulp \leq \frac{ulp(x)}{x} < 2 \times ulp$. C'est pourquoi un résultat analytique sur l'erreur relative ou l'erreur absolue est bien souvent insuffisant.

1.2 Propagation et Absorbtion des Erreurs d'Arrondi

Alors que la génération et la propagation des erreurs d'arrondi ont été étudiées en détail pour l'addition flottante [Pic 76], on ne trouve pas un travail comparable pour la multiplication flottante. Des auteurs ont analysé la fiabilité des résultats d'une suite de multiplications dans [MM 73], mais la multiplication est bien adaptée au mode de calcul flottant et les algorithmes qui présentent un comportement original ou chaotique sont relativement rares. En utilisant un modèle probabiliste, nous allons vérifier que les chances que des erreurs successives se compensent dans une chaîne de multiplications sont relativement élevées. Nous étudierons ensuite une expression simple à base de multiplications dont la précision dépend de l'ordre d'évaluation des opérations élémentaires de multiplication.

1.2.1 Travaux Antérieurs

Nous avons vu dans [Cod 73, Mul 89, FM 90] entre autres, que l'ordre d'évaluation des termes d'une expression en arithmétique flottante, même normalisée, peut faire varier énormément la précision du résultat final. Des études ont permis de construire plusieurs familles de problèmes au

comportement chaotique. La précision du résultat dépend pour ces expressions à la fois de l'ordre d'évaluation et des détails de l'implantation de l'unité arithmétique si elle n'est pas normalisée.

Contrôler la précision de l'évaluation même des expressions simples est un problème important de l'arithmétique des ordinateurs. De nombreux programmes utilisent le calcul flottant pour obtenir un résultat. Si dans bien des cas, on peut supposer sans risque que le résultat est correct, cela n'est pas possible dès que des vies humaines ou des investissements importants dépendent de la fiabilité du résultat. Il faut alors analyser en profondeur le comportement du système et définir exactement les conditions fiables d'utilisation du programme. C'est ainsi que 28 soldats américains ont payé de leur vie l'utilisation inadaptée des missiles de défense Patriot pour une période prolongée au delà des spécifications du constructeur [USA 92].

Dans [FM 90], les auteurs proposent sur un mode ludique un algorithme simple basé sur la définition de la série exponentielle (voir Figure 1.11). Pour $n = 25$, la valeur du résultat devrait être $X_{25} \approx 0.04$. Une brève analyse mathématique montre que la suite X_n converge vers 0 quand n tend vers $+\infty$. Mais cette expression calculée à l'aide d'une calculatrice de poche donne $X_{25} \approx 1.2 \times 10^9$. La calculatrice utilisée par les auteurs était une Casio FX702P. Si nous l'évaluons maintenant avec un ordinateur compatible avec la norme IEEE cela donne $X_{25} \approx -1.4 \times 10^{14}$. La plus petite erreur de calcul a des retombées catastrophiques ! Ainsi, comme on ne peut pas utiliser une valeur exacte de $e - 1$ au départ, le processus est toujours divergent. Le signe du résultat final est déterminé par l'erreur initiale sur e et par les premiers pas de l'algorithme où les erreurs peuvent encore se compenser partiellement.

$X_0 := e - 1$
For $i := 1$ to n do
$\quad X_i := i \times X_{i-1} - 1$

FIG. 1.11 - *Suite "Exponentielle"*

Une autre suite est proposée dans [Mul 89]. Cette suite est définie par récurrence par une fonction continue qui possède trois points fixes : $\lambda_0 = 5$, $\lambda_1 = 6$ et $\lambda_2 = 100$. La limite de la suite dépend donc des conditions initiales. La suite proposée Figure 1.12 avec ses valeurs initiales converge vers λ_1. Mais λ_1 est répulsif alors que λ_2 possède un très grand bassin d'attraction. Ainsi, une petite erreur peut faire sortir la suite du bassin de λ_1. Dans ce cas, l'erreur est d'autant plus grave que la suite Y'_n ainsi obtenue converge très rapidement vers λ_2. L'utilisateur aura ainsi tendance à croire que cette convergence rapide est signe du bon fonctionnement de la méthode.

$Y_0 := \frac{11}{2}$ et $Y_1 := \frac{61}{11}$
For $i := 1$ to n do
$\quad Y_{i+1} := 111 - \frac{1130}{Y_i} + \frac{3000}{Y_i Y_{i-1}}$

FIG. 1.12 - *Attracteurs Éloignés*

Les fonctions trigonométriques nous offrent un dernier exemple. Ces fonctions sont implantées

uniquement sur un intervalle du type $[-\pi, \pi]$ ou $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Pour pouvoir calculer la valeur des fonctions trigonométriques sur l'ensemble du domaine flottant $\mathbb{F}$, l'unité flottante réduit l'argument à un nombre appartenant à l'intervalle spécifique. W. Cody a présenté dans [Cod 73] ses expériences avec le calcul de sin(22). Comme $\pi \approx \frac{22}{7}$, le mécanisme de réduction d'argument entraîne un déplacement de la précision et annule beaucoup des bits significatifs de l'argument. Par exemple, sur une TI 25 seuls deux chiffres décimaux du résultat sont corrects. Avec les ordinateurs modernes disponibles actuellement, la précision des fonctions élémentaires est plus grande [DMMM 95, FB 91], mais il est toujours possible d'obtenir un résultat peu précis par exemple pour sin(355). On vérifiera que le terme suivant $\frac{22}{7}$ du développement en fractions continues de π est $\pi \approx \frac{355}{113}$.

La propagation des erreurs dans les expressions numériques a été étudiée en détail dans le passé. Ainsi Michèle Pichat a consacré sa thèse au comportement de l'addition flottante au regard de l'ordre d'évaluation des termes. Dans ses travaux, elle décrit plusieurs algorithmes en vue d'obtenir la plus grande précision possible d'une séquence d'additions flottantes [Pic 72]. Récemment, D. Priest a présenté le détail des algorithmes [Pri 91] nécessaires à l'implantation d'une bibliothèque de calcul multi-précision avec une arithmétique fidèle. La méthode des perturbations [LV 74] a été introduite pour de larges expressions qui rendraient difficile tout suivi explicite de l'erreur d'arrondi. Ces techniques sont utilisées dans des environnements capables de mettre en œuvre toutes les opérations flottantes et les opérations élémentaires. Certains résultats récents ont été présentés pour la méthode CESTAC dans [CV 92].

1.2.2 Modèle Probabiliste pour la Multiplication Flottante

Pour cette étude, nous allons utiliser un modèle probabiliste semblable à celui décrit dans [Che 88] pour modéliser les erreurs d'arrondi élémentaires dans les expressions complexes. Nous verrons comment des erreurs probabilistes peuvent être propagées ou absorbées au cours des calculs : on dit un peu faussement que les erreurs se compensent. Sur un ordinateur compatible avec la norme IEEE, la mutliplication $x = vw$ est implantée par l'opération flottante $p = v * w$ où $*$ vérifie la règle suivante si l'arrondi au plus près est le mode d'arrondi actif.

$$p = v * w = \circ(vw)$$

La modélisation stochastique usuelle consiste à introduire une variable aléatoire χ pour modéliser l'erreur d'arrondi par rapport à la valeur exacte. La nature de la variable χ dépend de la nature de l'opération, de la valeur du résultat mathématique et des spécifications de l'unité flottante.

$$p = vw + \chi_p \, ulp$$

Dans les faits, l'erreur χ représente l'erreur d'arrondi $\circ(vw) = vw + \chi_p \, ulp$. D'après les spécifications de la norme, si $p \in [1, 2[$, le résultat de l'opération flottante se trouve au plus à la distance $\frac{ulp(p)}{2}$ du résultat mathématique. Ainsi, le domaine de χ_p est inclus dans l'intervalle $[-\frac{1}{2}, \frac{1}{2}]$. Dès que la taille des mantisses dépasse la dizaine de bits, nous pouvons supposer à la suite de [FG 76] que la répartition de χ est uniforme sur l'intervalle. La multiplication agissant principalement sur les mantisses des opérandes, on peut aisément se ramener au cas où $v \in [1, 2[$, il vient alors que :

$$\frac{1}{2} < \frac{1}{v} \leq w < \frac{2}{v} \leq 2$$

Dans le cours d'un calcul, chacun des opérandes n'est pas connue exactement, mais l'on connait seulement sa représentation sous forme flottante. Ainsi, on n'effectue pas $p = v * w$, mais $P = \circ(\circ(v) \circ(w))$. La valeur de p peut être considérée comme correcte dans le du cas où :

$$p = \circ \left(\circ(v) \circ(w)\right) = \circ(vw)$$

Les arrondis $\circ(v)$ et $\circ(w)$ introduisent chacun une erreur représentée respectivement par les variables $\chi_v\ ulp$ et $\chi_w \frac{ulp}{\delta}$. Quand $w < 1$ il vient alors $\delta = 2$, sinon $\delta = 1$. Nous supposerons dans cet exposé que les variables χ_v, χ_w et χ_p sont indépendantes et réparties uniformément sur l'intervalle $[-\frac{1}{2}, \frac{1}{2}]$. Nous nous intéressons donc aux trois événements A, B, C qui suivent.

Évènement	Description
A	Les erreurs d'arrondi de v et w disparaissent dans l'opération
B	Une des deux erreurs se propage et apparaît dans le résultat
C	Le résultat calculé est distant de 2 ulp du résultat théorique

Évènement A Les deux erreurs d'arrondi sur les opérandes disparaissent si l'on vérifie :

$$\begin{array}{rc} & |\circ(v)\circ(w) - \circ(vw)| \leq \frac{ulp}{2} \\ \iff & \left|(v + \chi_v\ ulp)(w + \frac{\chi_w\ ulp}{\delta}) - (vw + \chi_p\ ulp)\right| \leq \frac{ulp}{2} \\ \iff & \left|w\chi_v + v\frac{\chi_w}{\delta} - \chi_p\right| \leq \frac{1}{2} \end{array}$$

On peut donc écrire la probabilité de l'évènement A comme suit.

$$\begin{aligned} P(A) &= P(\left|w\chi_v + \tfrac{v}{\delta}\chi_w - \chi_p\right| \leq \tfrac{1}{2}) \\ &= 1 - 2P(w\chi_v + \tfrac{v}{\delta}\chi_w - \chi_p \geq \tfrac{1}{2}) \end{aligned}$$

Le calcul se réduit donc au calcul du volume $V_{a,b}$ de l'intersection d'un cube de coté 1 et du plan défini comme suit. Afin de simplifier les calculs, on pourra se souvenir que a et b appartiennent à l'intervalle $[\frac{1}{2}, 2]$. Sans nuire à la généralité du problème, nous poserons de plus que $a \leq b$.

$$a * x + b * y - c \geq \frac{1}{2}$$

Nous avons obtenu la valeur de $V_{a,b}$ en validant nos calculs à l'aide du logiciel Maple. Nous noterons $\mu = \min(a, 2 - b)$, on obtient alors :

$$V_{a,b} = \frac{a^3 - 3a^2b + 3ab^2 + 3b^2\mu + 3b\mu^2 + \mu^3 + 6a^2 + 12ab - 12b\mu - 6\mu^2 - 12a + 12\mu}{48ab}$$

Pour retrouver la probabilité de l'évènement A présentée Figure 1.13, nous faisons la transformation suivante sur les données :

$$\begin{cases} a = \min(\frac{v}{2}, w) & \text{et} \quad b = \max(\frac{v}{2}, w) \quad \text{si} \quad w < 1 \\ a = \min(v, w) & \text{et} \quad b = \max(v, w) \quad \text{si} \quad w \geq 1 \end{cases}$$

□

Évènement C Grâce à une démarche comparable à celle que nous venons d'effectuer, on déduit que la probabilité de l'événement C se réduit à l'équation suivante.

$$P(C) = 2P(w\chi_v + \frac{v}{\delta}\chi_w - \chi_p \geq \frac{3}{2})$$

Il s'agit donc de calculer $V'_{a,b}$ qui est l'intersection d'un cube de coté 1 et du demi-espace défini par l'inéquation suivante.

$$a * x + b * y - c \geq \frac{1}{2}$$

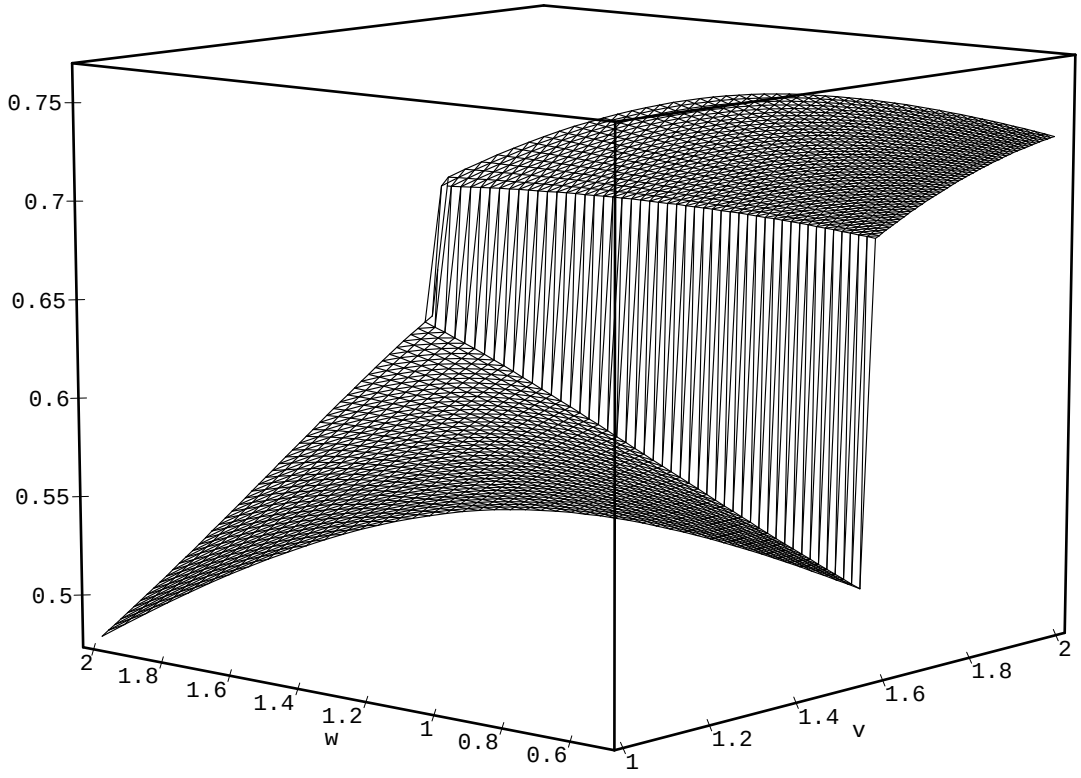


FIG. 1.13 - *Probabilité de Compensation Totale des Erreurs d'Arrondi*

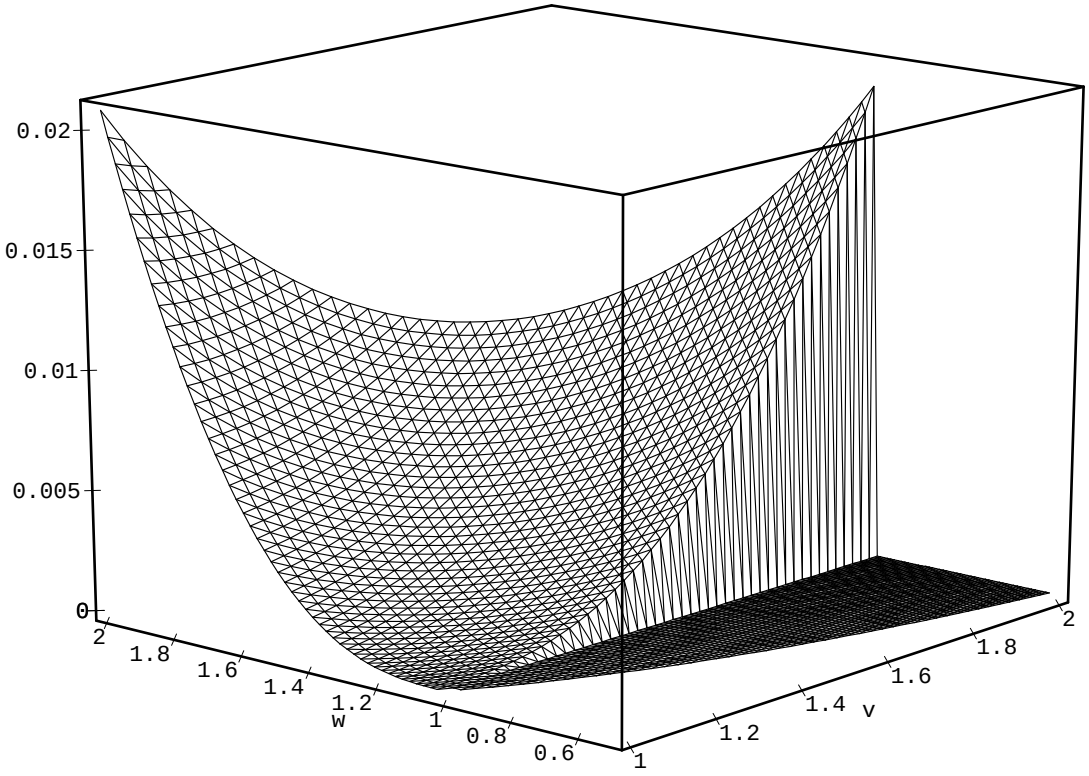


FIG. 1.14 - *Probabilité de Propagation des Deux Erreurs d'Arrondi*

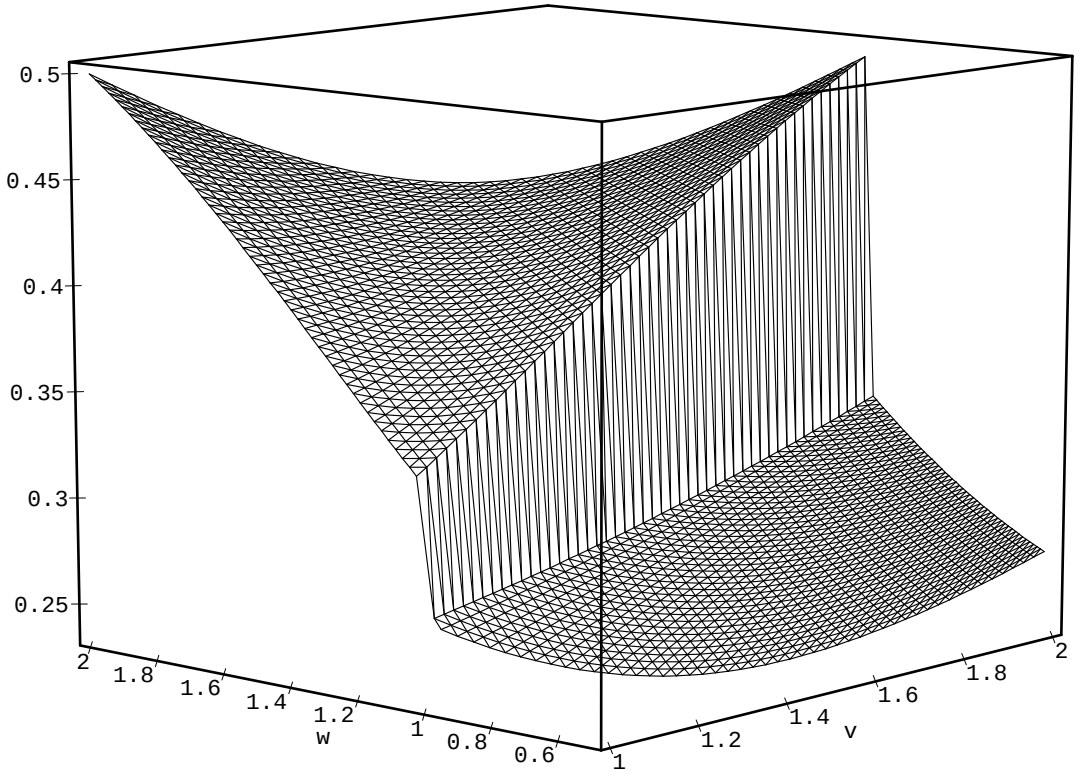


FIG. 1.15 - *Probabilité de Propagation d'une des Deux Erreurs d'Arrondi*

Avec les mêmes hypothèses que précédemment, on vérifie que $V'_{a,b}$ s'exprime ainsi pour $a+b \geq 2$:

$$V'a, b = \frac{(a+b-2)^3}{48ab}$$

On obtient la Figure 1.14 en utilisant le même changement de variable que pour l'événement A.

□

Événement B Les trois événements A, B et C constituent l'ensemble de l'univers possible. De ce fait, nous obtenons la probabilité de l'événement B présenté Figure 1.15 à partir des calculs qui ont été fait jusqu'ici.

□

On observe donc sur ces schémas que la probabilité que les deux erreurs se propagent est de l'ordre de 2%. Dans le même temps, on a entre 50% et 75% de chances, suivant la valeur des opérandes, que la multiplication se comporte comme si les opérandes qui lui ont été fournies etaient exactes, c'est à dire disponibles sans avoir été arrondies.

1.2.3 Perte de Précision dans une Multiplication Itérée

Parce que la multiplication est relativement bien adaptée au mode de calcul flottant, elle a été relativement ignorée jusqu'à présent. Nous étudions le comportement du calcul de l'expression u_n suivante d'après la valeur du paramètre n. On évalue cette expression sur un ordinateur compatible avec le format IEEE en utilisant les deux types normalisés disponibles — *single* et *double*. Le mode d'arrondi actif en machine est l'arrondi au plus proche.

$$u_n = \prod_{k=2}^{n} \left(1 - \frac{1}{k}\right) = \prod_{k=2}^{n} \left(\frac{k-1}{k}\right)$$

Par récurrence, on vérifie qu'en l'absence d'erreur d'arrondi, $u_n = \frac{1}{n}$ pour $n \geq 2$. L'étude expérimentale consiste à évaluer cette expression selon deux algorithmes différents : la méthode "croissante" C et la méthode "décroissante" D.

$$\begin{aligned} C_n &= \left(\ldots\left(\left(\left(\left(1-\tfrac{1}{2}\right)\times\left(1-\tfrac{1}{3}\right)\right)\times\left(1-\tfrac{1}{4}\right)\right)\ldots\times\left(1-\tfrac{1}{n-1}\right)\right)\right)\times\left(1-\tfrac{1}{n}\right) \\ D_n &= \left(\ldots\left(\left(\left(1-\tfrac{1}{n}\right)\times\left(1-\tfrac{1}{n-1}\right)\right)\times\left(1-\tfrac{1}{n-2}\right)\right)\ldots\times\left(1-\tfrac{1}{3}\right)\right)\right)\times\left(1-\tfrac{1}{2}\right) \end{aligned}$$

Pour chaque valeur de n, on construit les suites des produits partiels $C_p^{(n)}$ et $D_p^{(n)}$ présentées Figure 1.16. Si l'on suppose que chacune des opérations intermédiaires s'effectue sans erreur, on peut démontrer par récurrence que les suites ont pour valeurs respectives :

- $C_p^{(n)} = \frac{1}{p}$ avec $n \geq 2$ et $1 \leq p \leq n$;
- $D_p^{(n)} = \frac{n-p}{n}$ avec $n \geq 2$ et $0 \leq p \leq n-1$.

$C_1^{(n)} := 1$
For $p := 2$ to n do
$\quad C_p^{(n)} := C_{p-1}^{(n)} \times \left(1 - \frac{1}{p}\right)$

$D_0^{(n)} := 1$
For $p := 1$ to $n-1$ do
$\quad D_p^{(n)} := D_{p-1}^{(n)} \times \left(1 - \frac{1}{n+1-p}\right)$

FIG. 1.16 - *Méthodes d'Évaluation C et D — Croissante et Décroissante*

On vérifie ainsi que $u_n = C_n^{(n)} = D_{n-1}^{(n)}$. À la fin de chaque évaluation, on peut calculer l'erreur due à la méthode d'évaluation. On note pour le premier algorithme (C), $\Delta_n^C = C_n^{(n)} - u_n$ et pour le second algorithme (D), $\Delta_n^D = D_{n-1}^{(n)} - u_n$. Nous avons regroupé les valeurs de $\Xi_n^C = -\log|\Delta_n^C|$ et $\Xi_n^D = -\log|\Delta_n^D|$ pour des entiers n variant de 1 à 10^9 sur la Figure 1.17. Ces valeurs représentent l'ordre de l'erreur : comme il s'agit de l'erreur absolue, Ξ_n^C et Ξ_n^D compte le nombre de 0 à gauche dans l'écriture en virgule fixe de l'erreur. Les valeurs sont données pour des calculs effectués en double précision.

L'erreur relative $\delta_n = \frac{\Delta_n}{u_n}$ permet de calculer le nombre de bits exacts du résultat en virgule flottante $\xi_n = -\log|\delta_n|$. Les valeurs de ξ_n^C et ξ_n^D sont présentées Figure 1.19 pour des calculs en

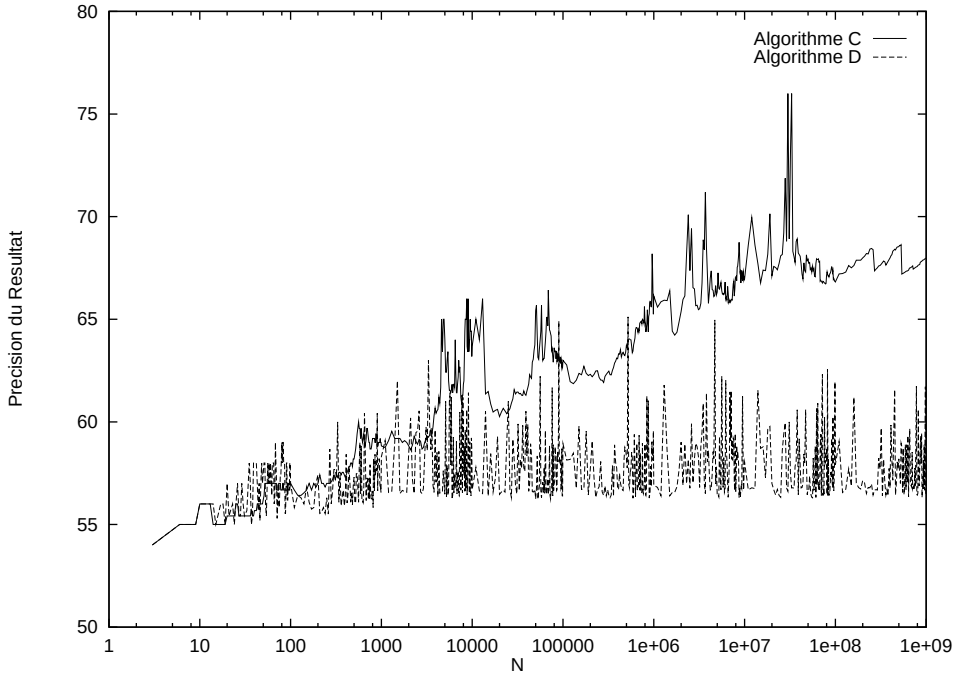


FIG. 1.17 - *Erreur Absolue des Algorithmes C et D en Double Précision*

double précision et Figure 1.18 pour les mêmes calculs en simple précision. La faible fiabilité de l'algorithme décroissant est nettement visible. Elle est encore plus marquante dans la Figure 1.20 où nous comparons le nombre de bits exacts des deux méthodes $\sigma_n = \xi_n^C - \xi_n^D$ pour les deux types disponibles.

Pour l'ensemble des graphiques, nous n'avons pas représenté toutes les valeurs de l'erreur pour n variant de 1 à 10^9. Pour chaque intervalle de la forme $[10^k, 10^{k+2}]$ nous utilisons uniquement les nombres de la forme $n = i \times 10^k$ où $i \in \{1 \ldots 100\}$. Les quelques points pour lesquels le résultat calculé était exact, c'est à dire $\Delta_n = 0$, ont été enlevés de l'échantillon pour ne pas encombrer le graphique.

En machine, on calcule donc les suites $\tilde{C}$ et $\tilde{D}$ de la façon suivante : toutes les données $1 - \frac{1}{p}$ sont normalisées de telle façon que $\overline{1 - \frac{1}{p}}$ n'entraîne pas de décalage vers la gauche ou vers la droite de $\tilde{C}$. À chaque fois que l'on effectue une multiplication, on modifie la valeur de la variable aléatoire $\Upsilon_{n,p}^C$ ou $\Upsilon_{n,p}^D$ qui représente la marche aléatoire de l'erreur.

$$\begin{aligned} \tilde{C}_p'^{(n)} &= \circ\left(\tilde{C}_{p-1}'^{(n)} \circ \left(\overline{1 - \tfrac{1}{p}}\right)\right) + \Upsilon_{n,p}^C \\ \tilde{D}_p^{(n)} &= \circ\left(\tilde{D}_{p-1}^{(n)} \circ \left(\overline{1 - \tfrac{1}{n+1-p}}\right)\right) + \Upsilon_{n,p}^D \end{aligned}$$

On définit la loi d'évolution de l'erreur en développant $\tilde{C}_{p+1}'^{(n)}$ et $\tilde{D}_{p+1}^{(n)}$. On ne traitera en détail que l'algorithme C. On introduit la variable aléatoire χ_p^C qui représente le nombre d'ulps que l'on perd dans les opérations d'arrondi. Pour calculer les caractéristiques de χ_p^C on utilise les travaux présentés à la section précédente.

$$\begin{aligned} \tilde{C}_{p+1}'^{(n)} &= \circ\left(\tilde{C}_p'^{(n)} \circ \left(\overline{1 - \tfrac{1}{p+1}}\right)\right) + \Upsilon_{n,p+1}^C \\ &= \circ\left(\left(\circ\left(\tilde{C}_{p-1}'^{(n)} \circ \left(\overline{1 - \tfrac{1}{p}}\right)\right) + \Upsilon_{n,p}^C\right) \circ \left(\overline{1 - \tfrac{1}{p+1}}\right)\right) \\ &= \circ\left(\tilde{C}_{p-1}'^{(n)} \circ \left(\overline{1 - \tfrac{1}{p}}\right) \circ \left(\overline{1 - \tfrac{1}{p+1}}\right)\right) + \circ\left(\Upsilon_{n,p}^C \circ \left(\overline{1 - \tfrac{1}{p+1}}\right)\right) + \chi_p^C \end{aligned}$$

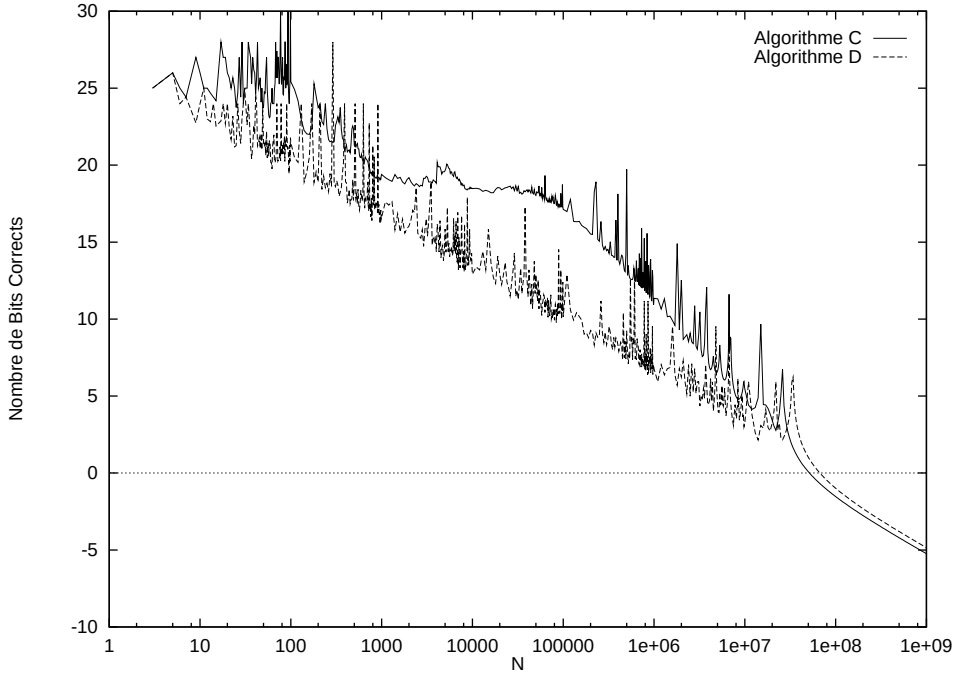


FIG. 1.18 - *Erreur Relative des Algorithmes C et D en Simple Précision*

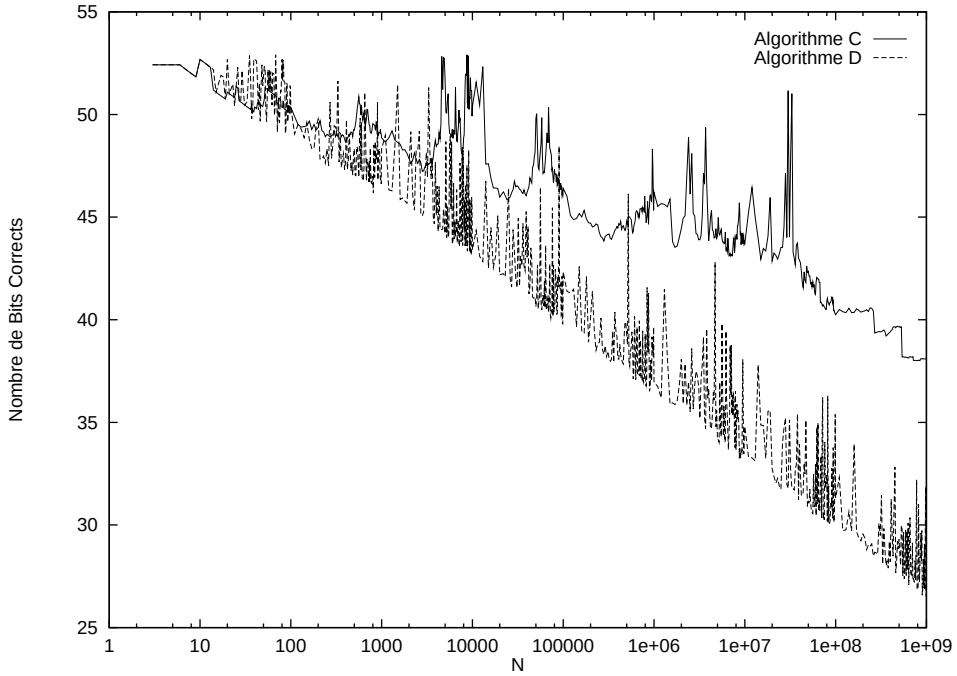


FIG. 1.19 - *Erreur Relative des Algorithmes C et D en Double Précision*

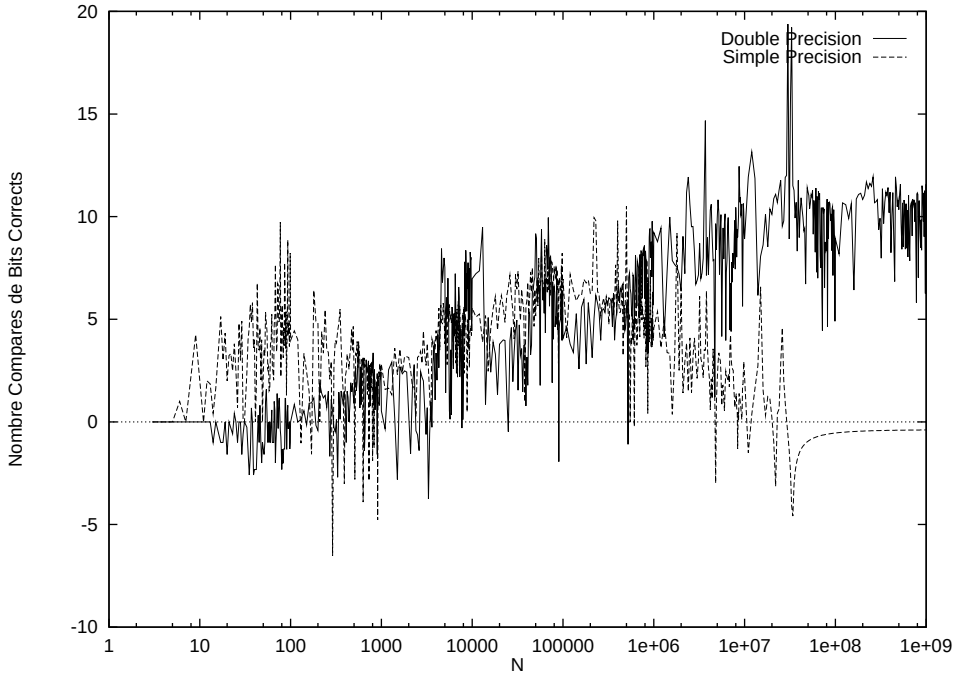


FIG. 1.20 - *Comparaison des Algorithmes C et D*

On constate que la progression de la marche aléatoire est fixée par la valeur des opérandes. Un rapide calcul nous montre que la modélisation de cette erreur seule n'est pas suffisante pour expliquer la différence de précision des deux méthodes de calcul. Il faut alors prendre en compte, comme le montrent les équations, le facteur d'échelle $\circ\left(\overline{1-\frac{1}{p+1}}\right)$ de la marche aléatoire.

1.3 Arrondi Fidèle

En machine, l'opération arithmétique $p = v \top w$ doit être obtenue comme si on calculait exactement le résultat $x = v \perp w$. On utilise ensuite le semi-morphisme d'arrondi $\square$ pour définir $p = \square(v \perp w)$. L'addition, la multiplication, la division et l'extraction de racine carrée sont implantées de manière à garantir l'exactitude du résultat. L'usage d'un bit de garde, d'un bit d'arrondi et d'un bit persistant [Gol 90] permet néanmoins de ne conserver que les chiffres significatifs du résultat et de ne pas travailler effectivement en précision infinie. Pour les fonctions élémentaires, sinus, cosinus, logarithme et exponentielle, on a plus souvent recours à un algorithme de calcul approché (approximation par un polynôme, CORDIC). Un tel algorithme renvoie le résultat approché x_0 connu avec une incertitude $\pm\delta$ suffisamment faible fixée par le type d'algorithme et les données de l'implantation. Le concepteur certifie que le résultat x de la fonction mathématique est dans l'intervalle $[x_0 - \delta, x_0 + \delta]$.

Soit x le résultat de l'opération mathématique à implanter. On connaît une approximation x_0 de x à δ prés. Il est important de pouvoir arrondir tous les points de l'intervalle $[x_0 - \delta, x_0 + \delta]$ à une même valeur flottante $v \in \mathbb{F}$ pour calculer exactement le résultat de l'opération flottante. En effet, si tous les points de l'intervalle considéré peuvent être arrondis à la même valeur flottante $v \in \mathbb{F}$, on connaît alors avec toute la précision nécessaire le résultat de l'opération flottante. Dans le cas contraire il n'est pas possible de donner le résultat correct de la fonction spécifiée par l'arrondi normalisé IEEE sans effectuer des calculs supplémentaires. Soit un intervalle $[a, b]$ de $\mathbb{R}$ de petite

taille. On s'intéresse à un nombre réel $x \in [a, b]$, et on désire savoir dans quelles conditions la connaissance de a et b est suffisante pour calculer $\Box(x)$. On ne fera aucune hypothèse sur x_0 si ce n'est que $x_0 \in [a, b]$. Dans l'énoncé précédent, il s'agit du centre de l'intervalle, mais on supposera ce point arbitraire dans le reste du développement.

Dans le cas général, un résultat seul sur la largeur de l'intervalle n'est pas suffisant. Soit I l'intervalle $[1-\delta, 1+\delta]$. Il ne peut pas être arrondi à une seule valeur avec un arrondi dirigé — $\bowtie$, $\triangle$ ou ∇. Ce résultat reste vrai aussi petit que soit $\delta > 0$. Si le mode d'arrondi actif est l'arrondi au plus près, l'intervalle $I = [1 + 1^+ - \delta, 1 + 1^+ + \delta]$ ne peut pas être arrondi correctement pour les mêmes raisons.

Nous avons introduit la quantité ulp afin d'obtenir des garanties analytiques sur les procédés d'arrondi, mais on ne peut décider directement si une approximation intermédiaire x_0, à δ près, est assez précise pour terminer le calcul interne. Il faut calculer explicitement la valeur des deux bornes a et b de l'intervalle de sécurité. Aucun des algorithmes actuellement utilisés pour le calcul des fonctions élémentaires [Mul 89, CW 80] ne fournit de garantie sur le positionnement des bornes de l'intervalle en un temps prédictible. En calculant des itérations supplémentaires, l'ordinateur ne peut qu'affiner légèrement la valeur de x_0 et réduire à chaque itération la taille δ de l'intervalle. Théoriquement, ce processus ne devrait s'arrêter que lorsqu'il est enfin possible d'arrondir l'intervalle courant à une seule valeur flottante. Grâce au théorème de Lindemann (1882) [Bak 75], nous pouvons garantir que le processus s'arrêtera pour un nombre relatif $v \in \mathbb{F} \subset \mathcal{Q}$, mais il n'est pas possible de borner le temps d'éxécution sur tout $\mathcal{Q}$.

Dans l'état actuel des connaissances les algorithmes de calcul des fonctions élémentaires ne peuvent pas être implantés conformément à la norme IEEE dans une unité flottante moderne. On ne peut garantir *a priori*, qu'au bout d'un nombre borné d'itérations, l'intervalle obtenu pourra être correctement arrondi à une seule valeur flottante. E. Swartzlander et M. Schulte ont certifié [SS 93] leur implantation des fonctions trigonométriques circulaires en simple précision par une recherche exhaustive sur l'ensemble du domaine de calcul. Dans les cas critiques, ils ont ainsi pu changer légèrement l'algorithme pour obtenir à coup sûr, en temps fixé, le résultat correct. Cette approche très coûteuse en simple précision est impossible pour des types supérieurs — *double* ou *quad*.

Il est donc actuellement impossible de proposer aux utilisateurs une implantation des fonctions élémentaires compatible avec les idées de la norme. Nous allons présenter un mécanisme étendu d'arrondi qui reprend l'ensemble des concepts essentiels de la norme. Nous avons rappelé plus haut la notation usuelle utilisée pour les modes d'arrondi normalisés IEEE : $\bowtie$, $\triangle$, $\circ$ et ∇. Les mécanismes d'arrondi fidèle que nous présentons utilisent deux modes d'arrondi à la fois que nous noterons donc $\circ\nabla$, $\nabla\circ$, $\nabla\nabla$, etc... On peut s'assurer que cette notation n'est pas ambigüe car un semi-morphisme d'arrondi est équivalent à l'identité sur $\mathbb{F}$. Deux nouveaux indicateurs d'état devront être ajoutés pour permettre à l'utilisateur ou à une bibliothèque de calcul de tester la validité du résultat et d'agir en conséquence. Si l'indicateur arrondi correct AC est actif, cela signifie que le calcul a été effectué conformément à la norme d'implantation. L'indicateur arrondi fidèle AF indique lui que la dernière opération a induit un arrondi correct ou un arrondi fidèle. Si aucun des ces indicateurs n'est actif, comme cela peut être le cas pour l'opération produit scalaire que nous avons définie par le passé [Dau 92], l'intervalle est trop grand et n'a pu être arrondi par aucun des mécanismes précis que nous proposons.

1.3.1 Arrondi Fidèle

Quand le mode d'arrondi au plus proche – négatif ($\circ\nabla$), est actif (voir Figure 1.21), le système tente dans un premier temps d'arrondir l'intervalle à l'aide de l'arrondi normalisé IEEE du mode

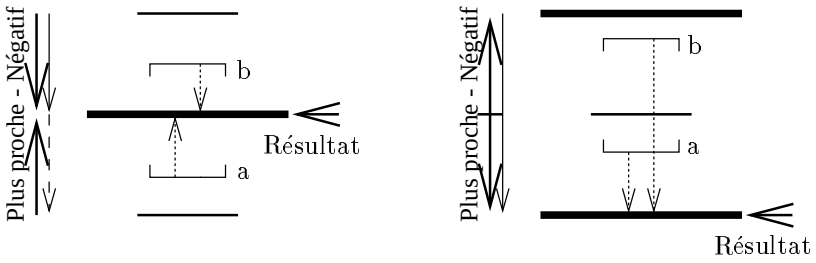


FIG. 1.21 - *Arrondi Plus Proche – Négatif*

primaire, c'est à dire l'arrondi au plus proche. Si c'est possible on obtient ainsi le résultat correct sans plus de calcul, les indicateurs AC et AF sont positionnés à 1. Dans le cas contraire le système tente d'arrondir l'intervalle à l'aide du mode secondaire, ici il s'agit de l'arrondi négatif. Si cette opération peut être réalisée, on obtient le résultat fidèle. Seul l'indicateur AF est alors actif. En cas d'échec l'intervalle ne peut pas être arrondi car il est trop imprécis. Le résultat de l'opération est spécifié séparément pour ce cas de figure avec l'implantation de l'opération comme c'est le cas pour le produit scalaire. Ni l'indicateur AC ni l'indicateur AF ne sont actifs.

$$\circ\,\nabla([a,b]) = \begin{cases} \circ(a) & \text{si} \quad \circ(a) = \circ(b) \\ \nabla(a) & \text{si} \quad \circ(a) \neq \circ(b) \quad \text{mais} \quad \nabla(a) = \nabla(b) \\ \text{Non défini} & \text{sinon} \end{cases}$$

Si $b-a < \frac{ulp(x_0)}{2}$, alors il y a au plus un nombre flottant $v \in \mathbb{F}$ dans l'intervalle $[a,b]$. Ce résultat n'est pas toujours suffisant pour arrondir correctement l'intervalle $[a,b]$. Par contre, si $b-a < \frac{ulp(x_0)}{4}$, l'intervalle est assez petit pour être arrondi, soit par l'arrondi au plus proche, soit par un mode d'arrondi dirigé (négatif ∇, positif $\triangle$ ou vers zéro $\bowtie$). Ainsi, pour garantir un arrondi fidèle, il suffit d'une approximation de largeur inférieure à $\frac{ulp(x_0)}{4}$.

On implante de la même façon les modes d'arrondi au plus proche – positif ($\circ\triangle$), et d'arrondi au plus proche – vers zéro ($\circ\bowtie$). Il est possible d'inverser l'ordre des modes actifs primaire et secondaire : le mode d'arrondi négatif – plus proche est présenté Figure 1.22. On construit aussi les modes $\bowtie\circ$ et $\triangle\circ$.

$$\nabla\circ([a,b]) = \begin{cases} \nabla(a) & \text{si} \quad \nabla(a) = \nabla(b) \\ \circ(a) & \text{si} \quad \nabla(a) \neq \nabla(b) \quad \text{mais} \quad \circ(a) = \circ(b) \\ \text{Non défini} & \text{sinon} \end{cases}$$

1.3.2 Arrondi Fidèle Dirigé

Nous venons de présenter un arrondi fidèle qui garantit que l'arrondi d'un intervalle suffisamment petit est l'arrondi mathématique précis de l'intervalle considéré en utilisant un mode prédéfini dans la norme IEEE. Le mode utilisé n'est pas totalement spécifié par l'utilisateur mais ce dernier conserve la maîtrise de l'arrondi car il définit les deux modes à utiliser.

Les modes d'arrondi vers zéro ($\bowtie$), d'arrondi négatif (∇) et d'arrondi positif ($\triangle$) sont appelés modes dirigés. Ils ont été définis pour permettre à l'utilisateur d'émuler une arithmétique d'intervalles. On peut ainsi, en contrôlant à chaque pas de calcul la direction de l'arrondi, certifier que le résultat obtenu est une approximation par excès ou par défaut du résultat exact.

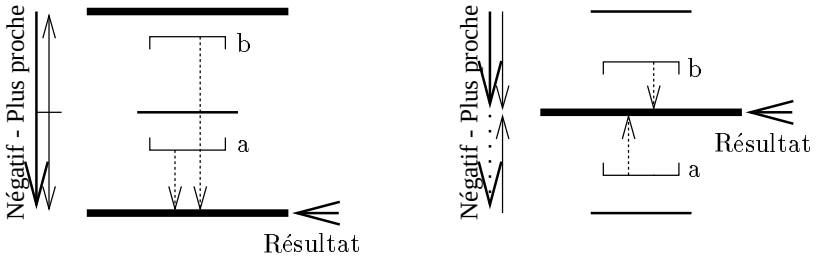


FIG. 1.22 - *Arrondi Négatif – Plus Proche*

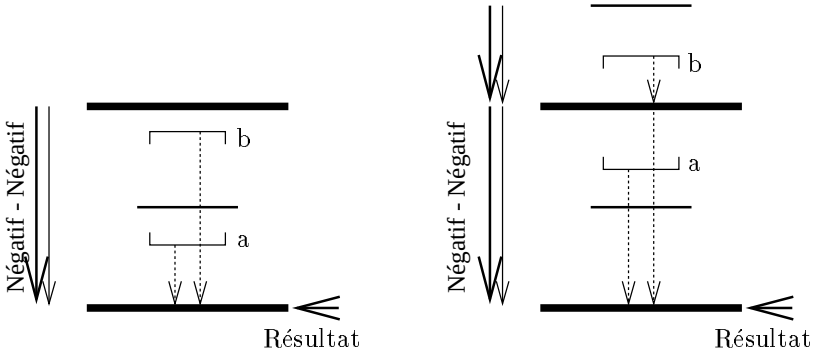


FIG. 1.23 - *Arrondi Négatif – Négatif*

Si les modes d'arrondi fidèle que nous venons de définir sont utilisés, l'utilisateur ne peut plus faire d'hypothèse sur le signe de l'erreur au terme de son calcul. Les modes d'arrondi fidèle dirigés que nous allons définir sont aussi précis que les modes d'arrondi fidèles présentés à la section précédente mais ils effectuent après l'arrondi un petit décalage d'un ulp du résultat pour conserver la propriété des modes IEEE d'arrondi dirigés.

Nous désignerons les modes d'arrondi fidèle dirigés par les symboles $\bowtie\bowtie$, $\triangle\,\triangle$ et $\nabla\,\nabla$. Il s'agit par exemple pour le mode d'arrondi vers zéro – vers zéro ($\bowtie\bowtie$), d'un arrondi vers zéro puis au plus proche. Dans le cas où l'arrondi au plus proche est utilisé, le résultat est déplacé d'un ulp vers zéro. On aurait pu désigner ce mode comme $\bowtie(\circ)\bowtie$. Un exemple du mode d'arrondi négatif – négatif est représenté Figure 1.23.

$$\nabla\,\nabla([a,b]) = \begin{cases} \nabla(a) & \text{si} \quad \nabla(a) = \nabla(b) \\ \circ(a)^- & \text{si} \quad \nabla(a) \neq \nabla(b) \qquad \text{mais} \quad \circ(a) = \circ(b) \\ \text{Non défini} & \text{dans les autres cas} \end{cases}$$

1.3.3 Arrondi Fidèle Non Déterministe

Les modes d'arrondi fidèle que nous avons définis sont à la fois déterministes et non déterministes. Sur une machine donnée, et avec un algorithme bien précis, le résultat sera toujours identique. Il n'y a pas de perturbation stochastique du résultat comme dans l'arithmétique perturbée [LV 74]. Mais, d'une machine à l'autre, ou d'une implantation des fonctions élementaires à une

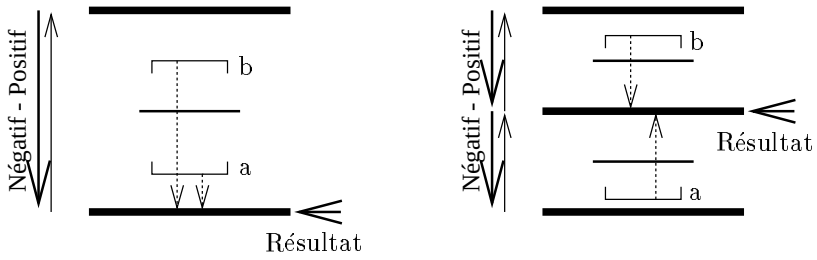


FIG. 1.24 - *Arrondi Négatif – Positif*

autre, il se peut que le résultat varie très faiblement. Les modes d'arrondi fidèles donnent l'arrondi normalisé IEEE du résultat mathématique et les informations d'état fournies par l'unité flottante, les indicateurs AC et AF, permettent à l'utilisateur de retrouver quel mode d'arrondi a été utilisé du mode d'arrondi primaire ou secondaire. Les modes d'arrondi fidèles dirigés ne renvoient pas nécessairement un résultat normalisé IEEE, mais dans ce cas, le biais introduit est ajouté seulement pour faciliter le traitement du résultat par l'utilisateur. Celui-ci peut, s'il le désire, consulter les indicateurs d'état et reconstruire le cas échéant le résultat normalisé de l'opération. Avec les modes d'arrondi fidèle non déterministes, l'utilisateur n'a plus aucun moyen de retrouver quel mode a été utilisé même si le résultat reste toujours un arrondi normalisé de l'opération mathématique exacte. On définit le mode d'arrondi positif – négatif ($\triangle\,\nabla$) comme suit. L'autre mode d'arrondi non déterministe est l'arrondi négatif – positif présenté Figure 1.24.

$$\triangle\,\nabla([a,b]) = \begin{cases} \triangle(a) & \text{si} \quad \triangle(a) = \nabla(b) \\ \text{Non défini} & \text{sinon} \end{cases}$$

Ce mode est beaucoup plus souple : on peut garantir l'arrondi d'un intervalle de taille $\frac{ulp(x_0)}{2}$. L'une des conséquence négatives de ce fait est que c'est le premier mode d'arrondi fidèle où l'on ne peut retrouver le mode d'arrondi normalisé utilisé pour calculer le résultat.

1.4 Arithmétique d'Intervalles

Valider les calculs numériques internes d'une application scientifique constitue une étape importante du développement. Les techniques couramment utilisées en machine font appel aux outils stochastiques [Che 88] et à l'arithmétique d'intervalles [Ble. 87]. Le bon usage de ces techniques nécessite une connaissance approfondie des logiciels disponibles, de leurs limites et de leur mise en œuvre. Comme aucune norme n'est disponible sur ces problèmes, chaque bibliothèque et chaque logiciel possède ses spécificités propres. Il est ainsi plus difficile de les appréhender et de choisir la solution la mieux adaptée au problème immédiat de l'utilisateur.

De plus, si le programmeur n'est pas suffisament informé des dernières nouveautés dans le domaine de la validation numérique, il est possible qu'il ne trouve pas le produit parfaitement adapté à son problème, ce qui l'amènera à modifier le code qu'il désire tester. Il y a alors un risque que l'opération de validation engendre de nouvelles erreurs dans le code qu'elle est censée valider. Le format que nous allons présenter ici permet de stocker un intervalle dans le même espace mémoire qu'un nombre flottant. Nous pouvons ainsi proposer à l'utilisateur de passer de l'arithmétique flottante normalisée à l'arithmétique d'intervalles en quelques instants. De plus, si

cette arithmétique venait à être réalisée sur circuit intégré, valider un code ne nécessiterait même pas de recompiler son programme.

On peut implanter, sur ce format de codage des intervalles, toutes les opérations flottantes: comme pour l'arithmétique flottante, il suffit, de calculer explicitement ou virtuellement les bornes de l'intervalle et de réduire par un semi-morphisme de compression l'intervalle au format de données fixé par le type courant.

1.4.1 Format de Données

Les intervalles sont représentés à partir d'une valeur flottante que l'on appelera leur origine. afin d'intégrer à l'arithmétique d'intervalles les idées qui ont prévalues dans la norme IEEE le mode d'arrondi actif fixe la position de l'origine de l'intervalle dans notre représentation. Pour le mode d'arrondi au plus proche, l'origine est au centre de l'intervalle. Avec les modes dirigés, l'origine représente la borne inférieure ou supérieure de l'intervalle selon le mode d'arrondi. En arithmétique d'intervalles le mode d'arrondi actif a donc moins d'incidence sur la valeur du résultat que sur sa présentation.

Nous avons ajouté deux champs de données au format normalisé de codage des nombres flottants : la largeur de l'intervalle est codée dans le premier champ, le second champ retient la position du séparateur que nous allons définir dans les pages à venir. La mantisse de l'origine et la largeur de l'intervalle partagent le même espace physique pour le stockage. Dans [MK 85] les auteurs proposent de coder de la même façon les deux composantes d'une fraction rationnelle sur un même champ. Pour cela, ils introduisent un séparateur dont la position est stockée dans le champ suivant.

Cette méthode a été choisie pour son économie et parce qu'elle simule le comportement de l'arithmétique flottante usuelle. En effet, les bits utilisés pour le codage de la largeur de l'intervalle sont de plus en plus nombreux au fur et à mesure que l'erreur d'arrondi augmente. Cela correspond souvent à la partie inutilisable d'un nombre flottant noyée dans le bruit des erreurs d'arrondi.

La longueur en bits de l'intervalle flottant est identique à celle du format normalisé qui lui correspond dans la hiérarchie IEEE : un intervalle flottant double précision occupe 64 bits (8 octets). De ce fait, on n'a besoin d'aucun espace mémoire supplémentaire pour stocker les intervalles.

La définition d'un intervalle flottant met en jeu les cinq champs présentés Figure 1.25. Trois de ces champs sont communs aux intervalles flottants et à l'arithmétique flottante usuelle bien que leur taille puisse avoir changé. Nous avons d'autre part apporté un soin particulier à conserver les positions relatives des champs qui sont repris dans l'arithmétique d'intervalle. En conséquence, il est possible pour un utilisateur peu averti de lire un intervalle flottant comme s'il s'agissait d'un nombre flottant. L'erreur ainsi commise est relativement faible et si les mécanismes de validation ont été correctement utilisés cela ne pose pas de grave problème.

- Le bit de signe s et l'exposant e conservent les mêmes emplacements et les mêmes définitions. Nous verrons plus loin comment ils sont utilisés.

- La fraction f de l'origine flottante est stockée sur un champ de taille variable. On construit l'origine de l'intervalle à l'aide de la fraction, du bit de signe et de l'exposant.

- Un nouveau champ l_m appelé séparateur est stocké à la fin de l'espace qui est réservé à la fraction dans le format flottant normalisé. Cette variable compte le nombre de bits alloués au codage de la mantisse (la fraction précédée du 1 implicite).

- Les bits qui ne sont pas alloués à la fraction sont utilisés par le champ contenant l'erreur δ. L'interprétation exacte de l'erreur δ dépend du mode d'arrondi actif. La taille du champ

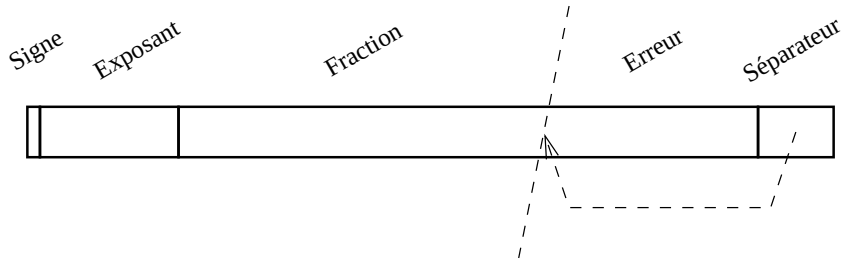


FIG. 1.25 - *Codage d'un Intervalle Flottant*

Type	Longueur (octets)	Signe	Exposant	Mantisse / Erreur	Séparateur
		(bits)			
Single	4	1	8	1 + 18	5
Single étendu	≥ 5	1	≥ 11	≥ 27	5 ou 6
Double	8	1	11	1 + 46	6
Double étendu	≥ 10	1	≥ 15	≥ 58	6 ou 7
Double étendu PC	10	1	15	1 + 58	6
Quad	16	1	15	105	7

FIG. 1.26 - *Hiérarchie des Intervalles Flottants*

l_m est ajustée automatiquement au cours des calculs pour représenter le plus fidèlement possible l'intervalle mathématique réel. Dans les 52 bits qui étaient originellement utilisés par la fraction dans le format IEEE, 6 sont utilisés pour contenir la position du séparateur afin que les champs fraction et erreur puissent avoir toutes les tailles possibles dans l'espace alloué.

La taille des différents champs d'un intervalle flottant est donnée Figure 1.26 pour l'ensemble des types normalisés.

1.4.2 Interprétation

L'interprétation d'un intervalle flottant est basée sur son point origine x_0. Avec le mode d'arrondi actif et la valeur du champ erreur, on peut définir exactement la position et la taille de l'intervalle relativement à l'origine x_0 (voir Figure 1.27). Trois des champs de l'intervalle flottant sont utilisés pour obtenir l'origine :

- La mantisse m, ou la mantisse entière M de x_0 est formée avec les bits du champ de fraction.
- Le séparateur l_m stocke la taille de la mantisse. D'après la définition du format, la longueur l_δ du champ d'erreur δ est directement connectée à l_m. Les deux valeurs sont conjuguées : $l_m + l_\delta = L$. L est donné pour chaque type dans la cinquième colonne de la Figure 1.26. On peut aussi définir l_m à partir de M.

$$l_m = \begin{cases} \lfloor \log M \rfloor + 1 & (M \neq 0) \\ 0 & (M = 0) \end{cases}$$

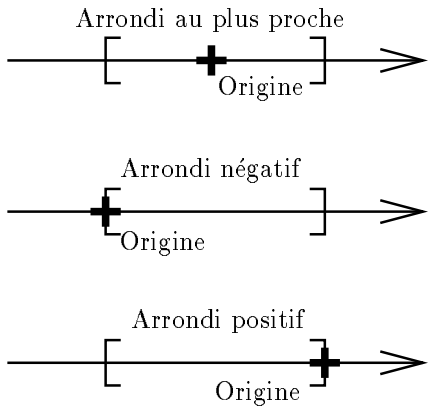


FIG. 1.27 - *Position de l'Origine dans l'Intervalle*

Arrondi Actif	Origine à partir de $[a, b]$	Intervalle à partir de x_0 et Δ
Positif	Borne Supérieure $\triangle(b)$	$[x_0 - \Delta, x_0]$
Au plus proche	Valeur Médiane $\circ(\frac{a+b}{2})$	$[x_0 - \frac{\Delta}{2}, x_0 + \frac{\Delta}{2}]$
Négatif	Borne Inférieure $\nabla(a)$	$[x_0, x_0 + \Delta]$

FIG. 1.28 - *Intervalle Flottant selon le Mode d'Arrondi*

– On obtient la valeur de l'origine avec les deux notations en utilisant l'exposant biaisé.

$$\begin{aligned} x_0 &= (-1)^s \times m \times 2^e \qquad (m = 1.f) \\ &= (-1)^s \times M \times 2^{e-l_m} \quad (M = 1 \oplus f) \end{aligned}$$

Nous nous intéresserons aux modes d'arrondi négatif, au plus proche et positif. L'origine de l'intervalle est obtenue en calculant l'arrondi par le mode actif de l'intervalle. L'intervalle représenté par le mot constitué des champs (M, δ, e) est interprété en fonction du mode d'arrondi. La taille absolue $\Delta = \delta \times 2^{e-l_m}$ de l'intervalle est donnée en fonction du champ d'erreur et de la valeur de l'exposant. À partir de l'intervalle $[a, b]$ on obtient l'origine x_0 et la largeur absolue Δ ; réciproquement on obtient l'intervalle à partir de ces deux valeurs (voir Figure 1.28).

Dans les applications réelles, les intervalles qui ont zéro pour origine sont très utilisés. Avec un mode d'arrondi actif au plus proche, ces intervalles représentent les situations où trop de bits ont été perdus au cours des calculs. Il est alors impossible d'obtenir le signe du résultat. On peut toutefois utiliser ces intervalles pour borner la valeur absolue du résultat. Si l'utilisateur travaille avec un mode d'arrondi dirigé, un intervalle dont l'origine est 0 contient une information de signe qui peut s'avérer très utile.

Si l'on s'en tient au mécanisme de codage des intervalles exposé plus haut, l'erreur absolue fait intervenir l'exposant de l'origine, il est alors impossible de représenter un intervalle non réduit à un seul point et dont l'orignie est 0. Dans la logique de la norme IEEE, nous pouvons stocker des intervalles qui ont pour origine zéro en attribuant à zéro l'exposant qui correspond à l'exposant biaisé nul soit $e_{min} - 1$. Par exemple, pour les nombres stockés en double précision, l'exposant

Origine				Erreur	
Mantisse	Exposant	Séparateur	Valeur	Codée	Absolue
1.345...	5	38	$1.345\ldots \times 2^5$	27	$27 \times 2^{5-38}$
0.000...	-2^{10}	20	$0\ldots \times 2^{-2^{10}}$	76	$76 \times 2^{-2^{10}-20}$
Non définie	6	0	$0\ldots \times 2^6$	146	146×2^6

FIG. 1.29 - *Exemples d'Intervalles Flottants en Double Précision*

est obtenu en retranchant 2^{10} à la caractéristique. De ce fait, l'exposant associé à zéro peut être établi comme -2^{10}. Il s'ensuit que l'interprétation du nombre flottant 0 sera dans les faits : $0 = (-1)^s \times 0 \times 2^{-2^{10}}$. Avec ces spécifications, la largeur maximale d'un intervalle ayant pour origine 0 est relativement restreinte : ainsi en double précision, on ne peut pas coder un intervalle plus large que $2^{-2^{10}+46} = 2^{-978}$.

Dans le format de codage que nous proposons, le nombre l_m indique la taille de la mantisse constituée du 1 implicite et de la fraction. La taille de la fraction seule est donc $l_m - 1$. Si l'on conserve le 1 implicite en tête de mantisse, la valeur $l_m = 0$ n'a pas de sens. Cela signifie donc que tous les bits de la mantisse, y compris le bit avant le point de fraction sont perdus. Nous nous autorisons ainsi à coder la quantité appelée zéro informatique [Vig 87]. L'origine de l'intervalle flottant est 0 puisque la mantisse ne comporte aucun bit. La valeur de l'exposant peut être quelconque et non plus seulement limitée à $e_{min} - 1$. On représente ainsi des intervalles de taille quelconque dont l'origine est zéro (voir Figure 1.29).

1.4.3 Semi-morphisme de Compression

L'implantation effective d'une arithmétique se compose d'un modèle de calcul et d'un modèle d'exécution. Le modèle d'exécution présente le comportement du circuit : dans la norme IEEE la clôture des opérations et la gestion des exceptions constituent une bonne part du modèle d'exécution. La hiérarchie de types et le semi-morphisme d'arrondi participent à la définition du modèle de calcul. Pour les intervalles flottants, la méthode de codage et le semi-morphisme de compression, qui joue le rôle de l'arrondi flottant, permettent de définir le modèle de calcul que nous mettons en place sur les intervalles flottants.

Le semi-morphisme d'arrondi détaillé dans la norme IEEE est un semi-morphisme de compression : il transfome n'importe quel nombre flottant de précision arbitraire en un nombre flottant de longueur fixée. Le nombre initial peut aussi bien avoir une écriture finie comme $1 + 2^{-k}$ où $k \in \mathbb{Z}$, ou infinie comme $\frac{1}{3}$. La compression entraine bien évidemment une perte d'information dès que le nombre n'est pas représentable exactement dans le format flottant en cours.

En définissant un semi-morphisme de compression qui s'applique à tout intervalle flottant de $\mathbb{R}^2$, on pourra implanter toutes les opérations mathématiques dans notre modèle de calcul : le résultat d'une opération en arithmétique d'intervalles est l'intervalle comprimé obtenu à partir du résultat exact en utilisant les données exactes. On définit ainsi des exigences de qualité comparables à celles requises par la norme d'implantation de l'arithmétique flottante.

Nous présentons les intervalles sous la forme d'un triplet (M, δ, e). Le mode d'arrondi actif parmi les trois modes utilisés arrondi positif, arrondi au plus proche et arrondi négatif sera noté $\square$. Appliqué à la mantisse entière, l'opération d'arrondi renvoie suivant le cas à l'entier le plus proche, à l'entier inférieur ou à l'entier supérieur. Pour tout entier relatif i, on définit l'origine $x_0^{(i)}$ comme

suit.

$$\begin{aligned} M^{(i)} &= \square\left(\tfrac{M}{2^i}\right) \\ l_m^{(i)} &= \begin{cases} \lfloor \log M^{(i)} \rfloor + 1 & \text{si} \quad M^{(i)} \neq 0 \\ 0 & \text{sinon} \end{cases} \\ e^{(i)} &= e - l_m + l_m^{(i)} + i \\ x_0^{(i)} &= (-1)^s \times M^{(i)} \times 2^{e^{(i)} - l_m^{(i)}} \end{aligned}$$

Pour tout i, on vérifie que la valeur $\delta^{(i)}$ donnée ci-dessous permet de construire l'intervalle minimal $I^{(i)}$ centré sur $x_0^{(i)}$ et contenant I.

$$\begin{aligned} R^{(i)} &= \begin{cases} |M - M^{(i)} \times 2^i| & \text{si} \quad \square = \nabla \text{ ou } \square = \triangle \\ 2 \times |M - M^{(i)} \times 2^i| & \text{si} \quad \square = \circ \end{cases} \\ \delta^{(i)} &= \left\lceil \tfrac{\delta + R^{(i)}}{2^i} \right\rceil \end{aligned}$$

Arrondi Négatif vers $-\infty$ Nous allons prouver pour le mode d'arrondi ∇ que l'intervalle $I^{(i)}$ contient I. De plus tout intervalle ayant pour origine $x_0^{(i)} \equiv (M^{(i)}, e^{(i)})$ et qui contient I doit contenir $I^{(i)}$.

Nous définissons les bornes de l'intervalle $I = [a, b]$ comme suit.

$$a = M \times 2^{e - l_m} \qquad \text{et} \qquad b = (M + \delta) \times 2^{e - l_m}$$

D'après la définition $I^{(i)} = [a^{(i)}, b^{(i)}]$, où :

$$a^{(i)} = M^{(i)} \times 2^{e^{(i)} - l_m^{(i)}} \qquad \text{et} \qquad b^{(i)} = \left(M^{(i)} + \delta^{(i)}\right) \times 2^{e^{(i)} - l_m^{(i)}}$$

Or $M^{(i)}$ vérifie la propriété suivante.

$$M - 2^i + 1 \leq 2^i \times M^{(i)} \leq M$$

On en déduit donc facilement que $a^{(i)} \leq a$. De plus, en substituant à $e^{(i)}$ et $M^{(i)}$ leur valeur dans l'expression de $b^{(i)}$, il vient après simplification :

$$b^{(i)} = \left(2^i M^{(i)} + 2^i \delta^{(i)}\right) 2^{e - l_m}$$

Et d'après la définition de $R^{(i)}$:

$$b^{(i)} = \left(M - \mathrm{R}^{(i)} + 2^i \delta^{(i)}\right) 2^{e - l_m}$$

Or $\delta^{(i)}$ vérifie :

$$e + \mathrm{R}^{(i)} \leq 2^i \left\lceil \frac{\delta + \mathrm{R}^{(i)}}{2^i} \right\rceil < e + \mathrm{R}^{(i)} + 2^i - 1$$

D'après la première partie de l'inégalité précédente on déduit la propriété suivante et donc finalement que $I^{(i)}$ contient I.

$$b^{(i)} \geq \left(M - \mathrm{R}^{(i)} + \delta + \mathrm{R}^{(i)}\right) 2^{e - l_m}$$

Construisons maintenant l'intervalle $I'^{(i)}$ comme suit. L'origine de l'intervalle est la même que celle de $I^{(i)}$, mais le champ d'erreur est changé en $\delta'^{(i)} = \delta^{(i)} - 1$. Il s'ensuit que l'intervalle est représenté par le triplet $(M^{(i)}, e^{(i)}, \delta'^{(i)})$. Nous allons vérifier que la borne supérieure $b'^{(i)}$ de l'intervalle n'est pas suffisante pour contenir tout l'intervalle initial I.

$$b'^{(i)} = \left(M^{(i)} + (\delta^{(i)} - 1)\right) \times 2^{e^{(i)} - l_m^{(i)}}$$

Comme précédemment, nous remplaçons $e^{(i)}$ et $M^{(i)}$ par leur valeur.

$$b'^{(i)} = \left(M - \mathrm{R}^{(i)} + 2^i\delta^{(i)} - 2^i\right) 2^{e-l_m}$$

D'après la seconde partie de l'inégalité que nous avons écrit sur $\delta^{(i)}$, il vient :

$$b^{(i)} < \left(M - \mathrm{R}^{(i)} + \delta + \mathrm{R}^{(i)}\right) 2^{e-l_m}$$

□

Arrondi Positif vers $+\infty$ Le mécanisme de la preuve pour l'arrondi positif est sensiblement identique à celui que nous venons de voir. Dans ce cas, la différence se fait sur la borne inférieure $a^{(i)}$ de l'intervalle.

□

Arrondi au Plus Proche La preuve présentée pour l'arrondi au plus proche est un peu différente des deux précédentes car elle dépend de la position initiale du centre. La borne critique qui permet de déterminer la valeur limite de $\delta^{(i)}$ ne peut être assignée en début de preuve. Suivant le signe de $M - 2^i \times M^{(i)}$, on considère la borne inférieure $a^{(i)}$ ou la borne supérieure $b^{(i)}$. On obtient par leurs définitions respectives la valeur de a et de b

$$a = \left(M - \frac{\delta}{2}\right) \times 2^{e-l_m} \qquad \text{et} \qquad b = \left(M + \frac{\delta}{2}\right) \times 2^{e-l_m}$$

Par une substitution identique à celle que nous avons effectuée plus tôt, on obtient :

$$a^{(i)} = \left(2^i \times M^{(i)} - 2^{i-1}\left\lceil \frac{\delta + \mathrm{R}^{(i)}}{2^i} \right\rceil\right) 2^{e-l_m}$$

Pour la borne supérieure, on obtient de la même façon :

$$b^{(i)} = \left(2^i \times M^{(i)} + 2^{i-1}\left\lceil \frac{\delta + \mathrm{R}^{(i)}}{2^i} \right\rceil\right) 2^{e-l_m}$$

Sans nuire à la généralité du résultat, on suppose que $M \geq 2^i \times M^{(i)}$. Cela signifie que M est arrondi par défaut pour i. Il faut donc vérifier que $b^{(i)} \geq b$, l'autre borne étant plus large. On écrit, avec une technique semblable à celle utilisée précédemment, la propriété suivante.

$$b^{(i)} \geq \left(M - \frac{R^{(i)}}{2} + \frac{\delta + R^{(i)}}{2}\right) 2^{e-l_m} = b$$

Pour finir, vérifions que la valeur de $\delta^{(i)}$ est minimale. On le déduit du fait que l'intervalle $[a'^{(i)}, b'^{(i)}]$ de diamètre $\delta^{(i)} - 1$ ne contient pas $[a, b]$ comme le montre le résultat suivant.

$$\begin{aligned} b'^{(i)} &= \left(2^i M^{(i)} + 2^{i-1}\left\lceil \frac{\delta + \mathrm{R}^{(i)} + 1}{2^i} \right\rceil - 2^{i-1}\right) 2^{e-l_m} \\ &< \left(M - \frac{R^{(i)}}{2} + \frac{\delta + R^{(i)} + 2^i}{2} - 2^{i-1}\right) 2^{e-l_m} \\ &< \left(M + \frac{\delta}{2}\right) 2^{e-l_m} \quad < \quad b \end{aligned}$$

□

Intervalle	Initial	Arrondi	Tronqué
Origine	1.111000×2^8	1.00×2^9	1.11×2^8
Erreur	11010×2^2	1×2^7	11×2^6
Borne Inférieure	1.011110×2^8	0.10×2^9	1.00×2^8
Borne Supérieure	10.010010×2^8	1.01×2^9	10.01×2^8

FIG. 1.30 - *Optimalité du Semi-Morphisme de Compression — Contre Exemple*

Pour une longueur L donnée, si on peut trouver une valeur de i où les longueurs de $M^{(i)}$ et de $\delta^{(i)}$ s'accordent exactement pour entrer dans le mot flottant, ce codage est optimal. Cela signifie que $I^{(i)}$ est le plus petit intervalle représentable dans le format de données et qui contienne I.

Si l'entier i est minimum tel que l'intervalle $I^{(i)}$ puisse être codé sur deux mots de longueur l_m et l_δ tels que $l_m + l_\delta \leq L$, alors $I^{(i)}$ est optimal sauf dans le cas suivant. L'exception a lieu quand le mode d'arrondi est l'arrondi au plus proche et que $M^{(i)}$ est arrondi à une une puissance exacte de 2. Nous présentons Figure 1.30 un contre-exemple de l'optimalité. La valeur initiale de M est donnée sur 6 bits. Nous ne pouvons conserver que 4 bits pour stocker la fraction et l'erreur. Dans ce cas particulier, à cause de la proximité d'une puissance exacte de 2, l'intervalle obtenu en troquant les derniers chiffres est plus précis.

Pour être en mesure de développer un circuit intégré qui réalise ces algorithmes, on sera probablement obligé dans un premier temps de calculer les deux bornes de l'intervalle. À partir des bornes, on peut sans difficulté réduire le triplet jusqu'à un triplet de taille normalisée avec l'algorithme de compression que nous allons spécifier cette section. Suivant la longueur de M et de δ, on ne doit essayer que deux valeurs pour i. On pose :

$$k = \left\lceil \frac{l_m + l_\delta - l}{2} \right\rceil$$

$$i_0 = \begin{cases} k & l_m > k, l_\delta > k \\ l_m - l & l_\delta < k \\ l_\delta - l & l_m < k \end{cases}$$

La Figure 1.31 présente les trois cas possibles de compression. Chaque cas entraîne un calcul différent de la valeur de i_0 comme nous venons de le présenter dans la formule précédente. Nous avons ajouté en grisé les retenues possibles qui pourront entraîner une étape de calcul supplémentaire où on aura recours à $i_0 + 1$. Soit $I^{(i_0)}$ est l'intervalle désiré, soit suite à une propagation de retenue, $I^{(i_0)}$ est trop grand et il s'ensuit que $I^{(i_0+1)}$ convient. Si aucune retenue n'engendre de déplacement de précision en atteignant une puissance de 2, les trois cas présentés dans la figure sont les seuls envisageables.

1.4.4 Calculer avec les Intervalles

En comparant l'arithmétique flottante usuelle et l'arithmétique d'intervalles que l'on vient de décrire, on s'aperçoit que les bits utilisés pour coder le séparateur sont effectivement perdus du point de vue de la précision. Néanmoins, cela représente uniquement 6 bits en double précision, soit approximativement 10% de la taille de la mantisse et une erreur relative de $2^{-44} \approx 5.68 \times 10^{-14}$.

Intuitivement, on peut penser que les bits qui servent à stocker le champ de l'erreur ne sont pas réellement perdus par rapport à ce que l'on pourrait attendre de l'arithmétique flottante usuelle.

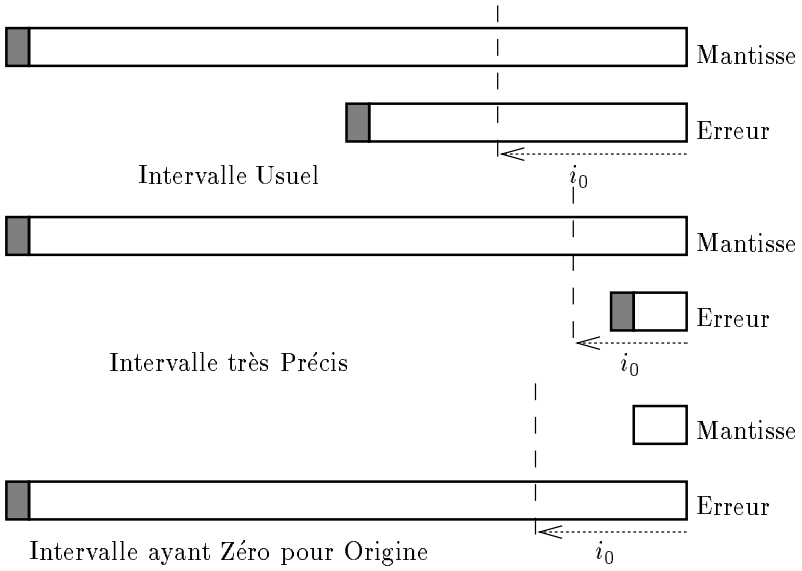


FIG. 1.31 - *Semi-morphisme de Compression*

On voit dans certains calculs que ces bits, qui sont utilisés pas l'unité flottante pour stocker une information non validée, contiennent énormément de bruit. Par une analyse très rapide qui exclut toute compensation des erreurs de calculs, on remarque que le champ d'erreur à tendance à croître avec le nombre d'opérations effectuées dans un calcul complexe en arithmétique d'intervalles de la même façon que le nombre de bits erronés à la fin de la mantisse augmente à cause du bruit généré par les erreurs d'arrondi et les décalages de précision.

Le codage de l'intervalle spécifié par notre arithmétique respecte le comportement attendu de l'arithmétique flottante. Plutôt que d'insérer des zéros ou de conserver des nombres sans signification à la fin du mot, on utilise ces bits pour stocker exactement la taille de l'intervalle. De ce fait, on ne peut pas considérer que les bits utilisés pour stocker δ soient vraiment perdus.

Si l'on construit un jour un circuit capable de réaliser les opérations usuelles sur les intervalles flottants ce circuit pourra prendre la place de l'unité flottante à chaque fois que l'utilisateur désirera valider ses calculs. Un code écrit originellement pour l'arithmétique usuelle pourra être immédiatement utilisé avec l'arithmétique d'intervalles. Un problème se posera avec les comparaisons : il faudra redéfinir la signification et la sémantique des tests tels que $a = b$, $a < b$, $a \leq b$, *etc...* Ce problème est semblable à celui que l'on rencontre avec une arithmétique perturbée [Che 88, Che 95] ; les solutions que nous proposons reprennent les idées qui ont été développées pour la méthode CESTAC. La comparaison de deux intervalles se fait en machine en calculant la différence de ces intervalles. Suivant le signe du résultat, on peut ordonner les intervalles. Si la comparaison fait apparaître un zéro informatique, nous nous trouvons dans le cas d'intervalles non disjoints.

Grâce au mécanisme d'exceptions, la norme IEEE d'implantation de l'arithmétique flottante peut réagir avec souplesse aux opérations qui génèrent des erreurs. L'arithmétique d'intervalles peut de la même façon adapter son comportement face aux tests d'après les besoins courants de l'utilisateur. Les comparaisons dangereuses sont celles où les intervalles a et b ne sont pas distincts comme sur la Figure 1.32. Pour ces comparaisons, l'utilisateur peut décider d'utiliser le comportement par défaut de l'unité flottante qui se comporte comme si l'un des membres était *Not*

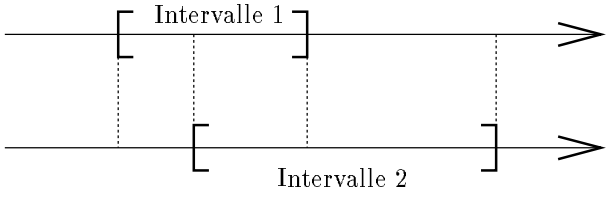


FIG. 1.32 - *Comparer deux Intervalles*

a Number en indiquant toujours que le test est faux. Il peut aussi, s'il le souhaite, mettre en place des techniques plus sophistiquées de choix et de propagation des incertitudes et des erreurs :

- L'arithmétique d'intervalles permet de tenir compte automatiquement dans les calculs de la précision des données. On pourrait ainsi arrêter le programme en cas de problème pour demander à l'utilisateur de donner les conditions initiales avec plus de précision.
- Si une exception est activée par une comparaison, le programme s'arrête et demande à l'utilisateur de lui indiquer le résultat de la comparaison. Si besoin est, l'utilisateur pourra être amené à effectuer une expérience physique afin de décider du résultat. Cette méthode ne peut être employée que dans la phase exploratoire d'un logiciel ou à des buts de recherche quand on désire trouver les endroits du logiciel où ont lieu des choix numériques instables.
- Le résultat est obtenu en comparant seulement les origines. Cela permet au programme de se terminer dans tous les cas sans erreur. Pour certifier le résultat, le programme établit dans le même temps une trace d'exécution. Cette trace reprend tous les choix dangereux effectués au cours de l'exécution. Il est de la responsabilité du concepteur de vérifier la pertinence de chaque choix, éventuellement par des mesures physiques. Si ce n'est pas possible, il faut alors transformer le programme pour réduire les sources d'instabilité numérique.

Le format que je viens de proposer pour l'arithmétique d'intervalles est très proche du format de la norme IEEE. On peut ainsi utiliser les codes existants sans modification — variable, structures de données, pointeurs, *etc.*..On peut également utiliser le résultat d'un calcul d'intervalles comme un nombre flottant pour l'affichage. Si l'utilisateur a vérifié que l'intervalle possède bien la précision requise, le fait de le lire comme un nombre flottant n'entraîne qu'une faible erreur. Il faut néanmoins offrir pour plus de précision de nouvelles fonctionnalités pour extraire les paramètres d'un intervalle ou former un intervalle à partir de ces paramètres :

Conversion Intervalle–Flottant Ces fonctions retournent la borne supérieure, la valeur médiane, la borne inférieure ou la largeur de l'intervalle.

Conversion Flottant–Intervalle Permet de construire un intervalle à partir de deux des paramètres précédents.

Adaptation des Intervalles Un ensemble de fonctions permettant, à partir d'un intervalle et d'un paramètre, de construire un nouvel intervalle plus large ou plus précis.

1.5 Prochains Pas

Avec le développement des ordinateurs, nous dépendons de plus en plus des simulations numériques. On a pu constater avec le problème de la division du Pentium que les ordinateurs per-

sonnels servent maintenant dans tous les domaines d'applications et que les utilisateurs ont aussi bien recours à l'arithmétique flottante qu'à l'arithmétique entière. Le calcul numérique n'est plus le domaine réservé de quelques scientifiques speécialisé dans l'analyse numérique.

La norme IEEE a fait ses preuves, mais avec la montée en puissance des ordinateurs proposés à un public de plus en plus large, on est arrivé aux limites de nos machines. Les technologies d'intégration et le développement des machines RISC ouvrent la possibilité de nouvelles fonctionnalités pour l'arithmétique flottante. La communauté scientifique doit s'interroger sur les priorités et guider la conception des unités arithmétiques de demain comme elle l'a fait lors du développement de celles d'aujourd'hui. Dans ces travaux, nous avons toujours gardé les points suivants présents à l'esprit.

- Pour être adoptée par les utilisateurs, une arithmétique nouvelle devra être relativement proche de l'arithmétique existante. Même si l'on peut demander à des utilisateurs expérimentés de travailler avec des arithmétiques originales, il faudra attendre que celles-ci aient fait leurs preuves pour voir l'ensemble des machines évoluer.

- L'implantation de ces fonctionnalités ne devra pas ralentir le processus de calcul ni être trop gourmande en place sur le circuit. La plupart des codes fonctionnent correctement avec l'arithmétique flottante usuelle. Et une fois que l'erreur est isolée, il est souvent aisé d'y remédier. Les utilisateurs sont donc enclins à penser que leur code est correct, et refuseront souvent de valider un code si le coût est trop élevé. Pour les analystes qui ont déjà repéré les instabilités de leur code, les solutions que nous venons de proposer ne seront probablement pas suffisantes, mais il faut leur donner les possibilité d'intégrer facilement leurs techniques de calcul à l'arithmétique normalisée.

L'objectif est donc de proposer un aménagement de l'arithmétique usuelle qui pourrait être repris par le plus grand nombre. Augmenter la précision de chaque opération élémentaire est toujours avantageux. Il est probable que les futurs développeurs consacreront une grande partie de l'espace qui sera alloué dans les nouvelles unités arithmétiques à une implantation en quadruple précision. Il ne faut néanmoins pas faire uniquement appel à une précision étendue pour valider les calculs. Alors que la précision maximale disponible sur les unités flottantes ira en grandissant, il sera de plus en plus coûteux d'utiliser une précision étendue sans raison. Il faudra donc ajouter des opérations pour évaluer avec suffisamment de flexibilité la précision nécessaire.

Chapitre 2

Arithmétique En-ligne

L'usage du calcul à virgule flottante est maintenant une habitude bien ancrée. Les unités arithmétiques modernes accomplissent correctement une bonne part des tâches qui incombent aux ordinateurs, mais pour certains problèmes l'arithmétique usuelle n'est pas adaptée. Les points que nous avons vus jusqu'à présent et les solutions que j'ai proposées ne couvrent qu'une partie des problèmes qui peuvent se poser à un utilisateur désireux d'effectuer des calculs numériques. Si certaines personnes se tournent vers des arithmétiques originales, c'est bien souvent pour obtenir leur résultat avec une précision accrue en conservant des temps de calcul acceptables. Certains concepteurs sont aussi amenés à opter pour une arithmétique non standard par souci d'économie, parce que toutes les fonctionnalités d'un processeur flottant complet ne sont pas toujours nécessaires pour des applications spécialisées. L'arithmétique à virgule flottante employée dans les machines actuelles n'offre aucune méthode de contrôle des erreurs. De plus, on ne peut que très peu faire varier la précision. Par opposition, l'arithmétique en-ligne permet de contrôler automatiquement et d'adapter au cours du temps la précision de calcul de chaque résultat intermédiaire. Ainsi, les efforts (temps de calcul et espace mémoire mobilisé) sont dirigés vers les opérations dont le résultat est sensible. Si le calcul est instable suite au conditionnement des données, le modèle de calcul en-ligne est capable de reconnaître les données qui doivent être affinées pour obtenir le résultat avec la précision demandée par l'utilisateur.

Avec les progrès dans les circuits reconfigurables logiques, on observe une demande en provenance des utilisateurs qui sont intéressés par la conception de leur propre circuit de traitement numérique [BM 94, MMP 95]. Ils recherchent une bibliothèque de procédures qui permettraient d'implanter aisément les fonctions arithmétiques usuelles. L'accélérateur matériel PeRLe mis au point au *Paris Research Laboratory* par l'équipe de J. Vuillemin [BRV 92] de la société *Digital Equipment Corporation* est une excellente plate-forme de mise au point et de test pour une telle bibliothèque. Cette carte est composée de 23 processeurs reconfigurables Xilinx XC3090 [Xil 94].

L'arithmétique série opère sur des nombres qui sont transmis un chiffre à la fois à travers un lien série. Par opposition, l'arithmétique usuelle ne commence pas une opération tant que tous les chiffres de tous les opérandes ne sont pas disponibles. Dans le calcul en-ligne [Erc 84], les chiffres des opérandes sont transmis en série, poids fort en tête; c'est le contraire dans l'arithmétique série usuelle. Les arithmétiques série et en-ligne sont couramment employées par les circuits spécialisés de traitement du signal. Par exemple, les concepteurs choisissent souvent des circuits arithmétiques série quand les liens de communication entre les opérateurs (liaison téléphonique...) ne sont pas suffisamment puissants pour permettre une liaison parallèle. On retrouve également des opérateurs série quand la taille du circuit est une donnée critique de l'implantation. Récemment, les techniques de calcul en-ligne en grande base ont été utilisées dans un microprocesseur à destination du grand public pour l'implantation des fonctions transcendantes [Lyn 95]. Plusieurs groupes de recherche ont développé des cellules capables d'implanter les fonctions arithmétiques élémentaires [LE 93, MRM 93, EMT 95] sur circuits logiques reconfigurables. Néanmoins, la difficulté de programmation de ces circuits et la complexité des algorithmes de calcul en-ligne font que ces techniques ne peuvent pas être mises en œuvre dans un environnement non spécialisé.

Je décris dans la seconde partie de ce chapitre une machine virtuelle de calcul en-ligne basée sur le modèle d'exécution à jeton étiqueté. Dans le courant des années 1970 de nombreux projets de machines à flot de données ont vu le jour. Le paradigme de programmation s'est concrétisé dans un article de J. Dennis paru en 1974 [Den 74]. Dès lors, plusieurs projets ont exploité cette idée en présentant des modèles de machines statiques dont on ne retiendra que les plus marquants: le *Static Machine DataFlow* du groupe de Dennis au *Massachussets Institute of Technology* [DBL 80], le projet NEC au Japon [THH 80], le projet Hughes [Gau. 85] et le LAU à Toulouse [CHS 80]. Les machines à jeton étiqueté, qui sont capables d'exécuter des programmes sans les restrictions

des machines statiques, sont apparues avec la *Manchester DataFlow Machine* conçue en Angleterre [GKW 85], le Sigma-1 au Japon [YSHK 85] et la *Tagged-Token Machine* du groupe de Arvind au MIT [AI 83]. Nos travaux sur un nouveau mode de calcul en-ligne sont motivés par les résultats que ces groupes de recherche ont obtenus avec leurs prototypes et plus particulièrement avec leurs compilateurs. Nous avons adapté notre modèle d'exécution à celui de ces machines, afin de pouvoir tirer profit de développements éventuels dans le domaine des machines hybrides [FE 93] et des progrès déjà acquis grâce aux compilateurs qui sont capables de générer du code destiné aux microprocesseurs usuels [FCO 90]. Nous présentons le modèle d'exécution de l'unité Puce qui pourrait dans le futur offrir aux ordinateurs les services d'un coprocesseur numérique évolué: elle est facile d'utilisation et capable d'évaluer des expressions complexes.

Nous allons décrire la conception et le protocole de tests adopté pour l'implantation de cellules d'addition, de multiplication, de division et d'extraction de racine carrée sur prédiffusés actifs. La première section rappelle les notions essentielles du calcul en-ligne. Nous présentons les algorithmes standards sous une forme synthétique nouvelle et les modifications adoptées en vue de leur implantation sur prédiffusés actifs. La section 2 offre une vue détaillée de l'accélérateur matériel PeRLe. Ce produit conçu par DEC est un prototype de développement. La documentation technique sur la carte elle-même et sur les outils de programmation est relativement succincte. La Section 3 présente les détails de l'implantation du noyau Nacel et de la bibliothèque d'opérateurs associée. Nous allons décrire Section 4 le modèle Puce. Enfin, la Section 5 est consacrée aux algorithmes d'addition et de multiplication ainsi qu'à l'analyse d'une simulation de calcul de transformée de Fourier discrète qui calcule jusqu'à 1000 chiffres décimaux du résultat.

2.1 Calcul En-ligne

Nous avons vu dans le chapitre précédent que l'arithmétique flottante qui est implantée sur les ordinateurs a été en grande partie dictée par les habitudes des utilisateurs. Dans cette optique, l'outil de calcul numérique devrait pouvoir s'adapter aux besoins de chaque étape du calcul. Ainsi, certains résultats intermédiaires seront calculés grossièrement alors que l'on pourra s'étendre sur les données critiques. Les systèmes de calcul à précision multiple sont capables de réaliser ce type de calcul, mais l'environnement ne permet pas de contrôler efficacement et automatiquement les besoins en précision. Le calcul en-ligne adapte de façon transparente pour l'utilisateur la précision de travail de chaque opération élémentaire. Un système de calcul en-ligne non seulement garantit la validité de tous les chiffres du résultat produit mais de plus, il isole les paramètres critiques des données.

2.1.1 Système BS

Le calcul en-ligne n'a probablement pas été plus répandu par le passé parce qu'il oblige les concepteurs à utiliser un système redondant d'écriture des nombres. De tels systèmes ont été proposés par A. Avižienis dans ses travaux [Avi 61]. Les nombres sont écrits en numération de position, mais on a recours à des chiffres différents de ceux que nous utilisons couramment. Ainsi, en base dix, le système redondant étendu ou *carry save* utilise tous les chiffres entre 0 et 9 ainsi qu'un chiffre A supplémentaire pour représenter 10. Le système minimal redondant symétrique utilise les chiffres entre 0 et 5 et les symboles $\overline{1}$ à $\overline{5}$ qui représentent les chiffres étendus de -1 à -5. Grâce à ces notations on peut limiter la propagation des retenues dans l'addition de deux nombres. Cela signifie que l'on peut effectuer une addition chiffre après chiffre de la gauche vers la droite.

La méthode générale d'addition des nombres redondants en base $\beta > 2$ décrites dans les travaux

d'A. Avižienis [Avi 61] ne peut pas être utilisée en base deux. D'autres algorithmes sont néanmoins disponibles dans ses travaux pour la base 2. Les deux systèmes naturels binaires redondants utilisent l'ensemble de chiffres $\{0,1,2\}$ et l'ensemble $\{\overline{1},0,1\}$. Le premier système porte le nom de *Carry Save* (CS). On pourra noter que les structures d'addition deux-en-trois fort répandues dans les multiplieurs qui sont capables d'effectuer une addition sans propagation de retenue utilisent au moins implicitement la notation CS. Le second système porte le nom de *Borrow Save* (BS) [CR 78].

Tout nombre réel $x \in \mathbb{R}$ peut s'écrire sous la forme de la somme infinie suivante de chiffres en base 2. On confondra souvent une représentation sous forme de suite double $(x_k^+, x_k^-)^n$ et le nombre réel homeomorphe que nous noterons plutôt X. Les bits x_k^+ et x_k^- constituent une représentation *Borrow Save* de X.

$$X = \sum_{k=-\infty}^{\infty} (x_k^+ - x_k^-)2^{-k} \quad \text{avec} \quad (x_k^+, x_k^-) \in \{0,1\}^2$$

Les termes de la somme de rangs négatifs sont presque tous nuls. Dans un souci de concision, on pourra adopter la notation suivante, si tous les chiffres d'ordre $k < -i$ sont nuls :

$$X = \sum_{k=-i}^{+\infty} (x_k^+ - x_k^-)2^{-k} = \left(\sum_{l=0}^{+\infty} (x_l^+ - x_l^-)2^{-l} \right) 2^i$$

Pour coder un chiffre du système BS sur ordinateur il faut deux bits x_k^+ et x_k^-. Ainsi, on utilise deux fois plus de bits que la notation binaire non redondante. En base 16, il suffit de 5 signaux binaires pour un codage redondant, et le sur-coût n'est plus que de 20%. Il est tout à fait possible d'exploiter une plus grande base, et d'utiliser correctement la puissance de calcul d'un microprocesseur conventionnel [Maz 93]. Le noyau Nacel et l'ensemble des développements que nous examinerons dans cette section utilisent le système BS. Nous verrons dans la suite de ce chapitre un exemple d'emploi d'une base intermédiaire. Nous utilisons pour le prototype logiciel de Puce la base 10 pour permettre une lecture aisée des résultats dans la phase d'exploration.

L'additionneur BS utilise comme cellule de base la cellule Plus-Plus-Moins (PPM) qui est dérivée de la cellule d'addition normale *Full Adder* (FA). Nous avons représenté le schéma logique d'une cellule PPM comparée à une cellule FA dans la Figure 2.1. La cellule PPM calcule la valeur des signaux binaires p et q en fonctions de a, b et c tels que :

$$2 \times p - q = a + b - c$$

Il est aisé de réaliser une cellule PPM à partir d'une cellule FA. Pour cela il suffit d'ajouter logiquement sur l'entrée c et la sortie q un inverseur comme nous l'avons fait Figure 2.2. Une cellule PPM peut se transformer en changeant le signe de toutes ses entrées et de toutes ses sorties en une cellule Moins-Moins-Plus qui est utilisée dans l'additionneur BS. On n'indiquera que les signes ou que la position des inverseurs. A. Guyot, Y. Herreros et J-M. Muller ont proposé dans leurs travaux une implantation de la cellule PPM qui utilise seulement 20 transistors en technologie CMOS [GHM 89].

L'additionneur que nous présentons Figure 2.3 a été obtenu à l'aide de l'algorithme d'A. Avižienis. Il est détaillé pour la première fois dans [GHM 89]. Pour un additionneur sur i chiffres, nous appelons t_k les sorties de la première ligne d'addition, en prenant bien soin de reporter la retenue pour le terme positif comme dans les égalités suivantes

$$2t_{k+1}^+ - t_k^- = a_k^+ + b_k^+ - a_k^-$$
$$2s_{k+1}^- - s_k^+ = t_k^- + b_k^- - t_k^+$$

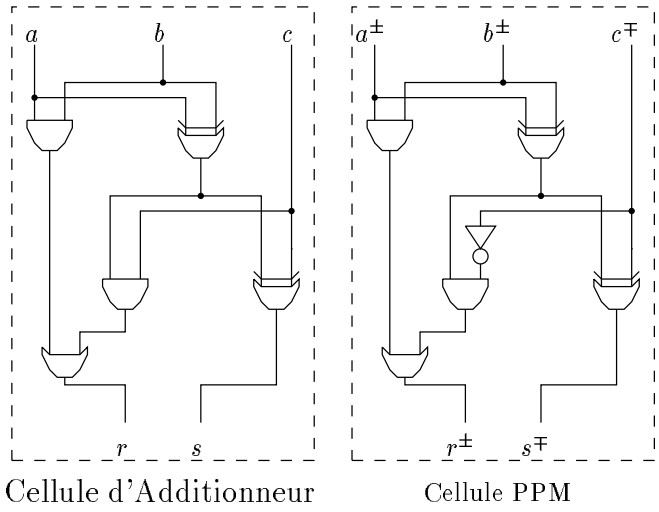


FIG. 2.1 - *Cellules d'Addition*

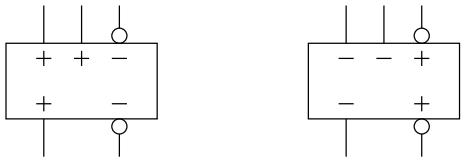


FIG. 2.2 - *Représentation des Cellules PPM*

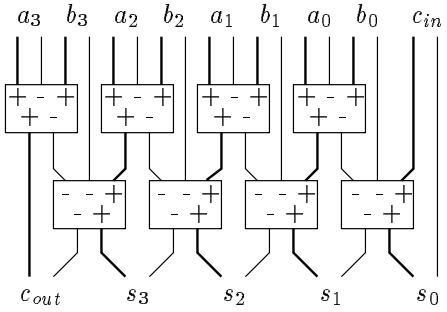


FIG. 2.3 - *Additionneur Borrow Save*

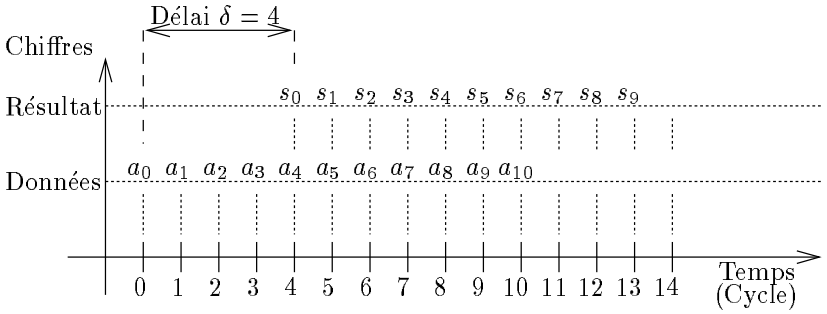


FIG. 2.4 - *Délai d'un Opérateur*

On vérifie aisément en posant $s^+_{i+1} = t^+_{i+1}$, $t^+_0 = c^+_{in}$ et $s^-_0 = c^-_{in}$ que

$$\sum_{k=0}^{i+1}(s^+_k - s^-_k)2^k = \sum_{k=0}^{i}(a^+_k - a^-_k)2^k + \sum_{k=0}^{i}(b^+_k - b^-_k)2^k + c^+_{in} - c^-_{in}$$

2.1.2 Principe de Fonctionnement

Avec un opérateur ou une unité de calcul en-ligne, on n'effectue pas un calcul en une seule étape sur la totalité des opérandes. Au contraire, à chaque cycle, le système fournit un nouveau chiffre des opérandes à l'opérateur. Pendant les premiers cycles, l'état de l'opérateur est construit et aucun résultat n'est disponible. Mais passé ce délai, l'opérateur produit un chiffre du résultat à chaque cycle. Le système donne des chiffres des opérandes à l'opérateur tant qu'il n'a pas obtenu le nombre de chiffres du résultat qu'il désire. Ce mécanisme est illustré Figure 2.4.

Avec l'écriture des nombres que nous venons de définir chaque nombre n'a pas un unique représentant. Nous utilisons une notation redondante et chaque nombre à l'exception de 0 admet plusieurs décompositions différentes. À partir d'une décomposition donnée on définit comme suit la valeur tronquée après le terme d'ordre p:

$$Tronc_p(X) = \sum_{k=-i}^{p}(x^+_k - x^-_k)2^{-k}$$

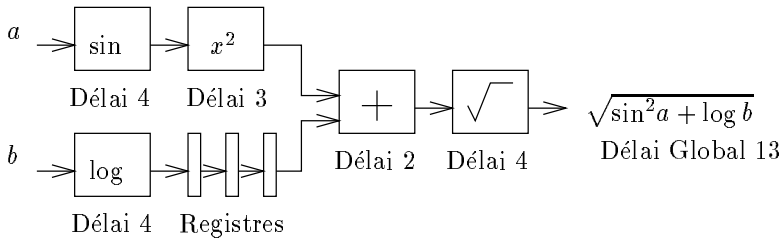


FIG. 2.5 - *Environnement En-ligne pour un Pipeline au Niveau du Chiffre*

On associe l'opérateur de calcul $P = X \top Y$ à l'opération mathématique $x \perp y$. L'opérateur est dit en-ligne, s'il existe une constante δ indépendante des données dans un domaine spécifique tel que pour tout k l'opérateur calcule le chiffre $p_k = p_k^+ - p_k^-$ à l'aide uniquement des valeurs de $Tronc_{p+\delta}(X)$ et de $Tronc_{p+\delta}(Y)$. On appelle délai d'un opérateur spécifique la valeur minimale de la constante δ pour son implantation. Le délai intrinsèque d'une fonction est la valeur minimale que peut avoir le délai d'un opérateur qui calcule cette fonction.

Le paradigme de calcul en-ligne est plus naturel car il correspond à notre démarche de pensée: l'utilisateur peut se faire rapidement une idée d'un résultat même s'il faut plus de travail pour affiner les calculs. Avec un opérateur de calcul en-ligne tel que l'addition en base 2, dès le troisième cycle, on commence à obtenir des chiffres du résultat. Si le résultat de la première opération est utilisé pour d'autres calculs, la seconde opération peut commencer dès que le premier chiffre du résultat de la première opération est disponible. Pour ces raisons, M. Ercegovac a présenté le modèle d'exécution des opérateurs en-ligne comme un *pipeline* au niveau du chiffre. Il s'agit d'un parallélisme à grain plus fin que les *pipelines* disponibles sur les ordinateurs actuels qui travaillent au niveau du mot. Nous avons représenté Figure 2.5 le chaînage de cinq opérateurs de calcul. Le délai de chaque opérateur est indiqué. Le temps de calcul n'est pas la somme des temps de calcul de chacun des éléments de la chaîne. Seuls les délais des différents opérateurs sont ajoutés.

En arithmétique en-ligne, il est possible de travailler avec une base quelconque. Le délai de chaque opérateur, qui est le temps d'initialisation de son état interne dépend légèrement de la base. Ainsi pour une grande base, le délai des opérateurs diminue un peu. Mais l'information requise pour initialiser un opérateur est invariante même si on augmente la base. Avec une grande base, il faudra seulement moins de cycles pour transférer toute l'information à l'opérateur. Pour l'utilisateur, les chiffres qui servent à l'initialisation sont toutefois perdus. Par exemple, quelle que soit la base, pour obtenir dix chiffres exacts du résultat avec un opérateur de délai trois, il faudra fournir à la machine treize chiffres des opérandes. On voit ainsi que l'on a tout intérêt à utiliser une base relativement petite.

2.1.3 Opérateurs Élémentaires

Des opérateurs de calcul en-ligne ont déjà été définis par le passé [BDKM 94, Erc 84, Kla 93, TE 77]. Certains des opérateurs sont optimisés pour calculer vite, d'autres pour réduire la taille du circuit. Plusieurs des auteurs cités ont tenu à présenter les algorithmes indépendamment de la base de numération. Afin de clarifier autant que possible ce tour d'horizon de l'arithmétique en-ligne et d'offrir au lecteur un ensemble synthétique simple d'équations, nous nous restreindrons uniquement à la base 2 et à la notation BS.

Nous présentons pour la première fois les algorithmes sous une forme où le délai est nul. Pour

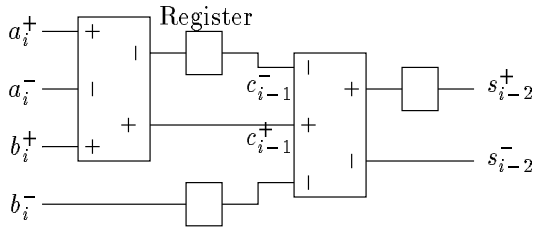


FIG. 2.6 - *Additionneur En-ligne*

obtenir un délai 0 avec tous les opérateurs, nous faisons au besoin apparaître des 0 en tête de mot. J-C. Bajard et J-M. Muller ont défini par le passé des opérateurs de délai 0 en modifiant les arguments [BM 92], mais ils n'ont pas décrit de nouveaux opérateurs. L'ajout d'un ou de plusieurs chiffres nuls en début de mot ne pose pas de problème important : si besoin est, on peut utiliser un opérateur de pseudo-normalisation. Par cette technique, les opérateurs de calcul en-ligne ont tous un contrôle très simple sans temps d'attente, à l'exception de l'opérateur de pseudo-normalisation, qui lui fait intervenir un temps d'attente variable mais borné. Pour mémoire, la normalisation d'un nombre ne peut être effectuée en-ligne car cela équivaut à convertir le nombre en notation usuelle non redondante. L'opération de pseudo-normalisation garantit que le premier chiffre du nombre donne un ordre de grandeur relativement précis de ce nombre. Cette forme est adaptée au contrôle limité que nous avons implanté sur la carte PeRLe. Elle ne le serait probablement pas dans le cas de la conception en VLSI d'un groupe de circuits en-ligne capables de travailler de façon asynchrone.

Nous présentons pour chaque opérateur les équations de récurrence et les invariants de boucle qui permettent de vérifier la validité des algorithmes.

Addition Nous pouvons déduire l'additionneur en-ligne du schéma de l'additionneur parallèle présenté plus haut. Avec un peu d'habitude, on découpe l'additionneur parallèle selon l'indice du chiffre, et on obtient le schéma Figure 2.1 qui contient deux cellules PPM et trois registres.

Pour définir l'opérateur d'addition par récurrence, nous avons besoin de la fonction signe étendue définie comme suit :

$$sign(x) = \begin{cases} 1 & \text{si} \quad x \geq \frac{1}{2} \\ 0 & \text{si} \quad x \in]-\frac{1}{2}, \frac{1}{2}[\\ -1 & \text{si} \quad x \leq -\frac{1}{2} \end{cases}$$

Il n'est pas possible de calculer en-ligne le signe exact d'un nombre. Le signe étendu lui non plus ne peut pas être calculé en ligne. Par contre, nous utiliserons $sign_i(X)$ pour représenter la composition $sign_i(X) = sign(Tronc_i(X))$ qui, elle, peut-être calculée dans un délai fini.

En début de calcul, l'ensemble des registres est positionné à 0. Cela signifie que X_0, Y_0, W_0 et S_0 sont tous nuls. Pour les opérandes X et Y, l'ensemble des chiffres d'ordre $j \leq 0$ sont nuls. Par exemple, pour X, on vérifie que $X_0 = Tronc_0(X_0) = 0$. Ces conditions impliquent que $X \in]-1, 1[$ et $Y \in]-1, 1[$. On vérifie aussi que sans autre restriction supplémentaire la récurrence suivante

calcule correctement le résultat. Il s'agit de l'algorithme correspondant à la Figure 2.6.

$$\begin{array}{ll} X_j & = X_{j-1} + x_j 2^{-j} \\ Y_j & = Y_{j-1} + y_j 2^{-j} \\ s_j & = sign_2(2W_{j-1}) \\ W_j & = 2W_{j-1} - s_j + \frac{x_j+y_j}{16} \\ S_j & = S_{j-1} + s_j 2^{-j} \end{array}$$

Les invariants de boucle sont :

$$S_j + W_j 2^{-j} = \frac{X_j + Y_j}{16} \quad \text{et} \quad W_j \in \left]-\frac{7}{8}, \frac{7}{8}\right[$$

Une simple substitution de W_j, S_j, X_j et Y_j par leur valeur en fonction de W_{j-1}, S_{j-1}, X_{j-1} et Y_{j-1} nous montre que le premier invariant peut être démontré par récurrence. Pour vérifier le second invariant, nous allons étudier trois cas selon la valeur de s_j. Dans le cas où $s_j = 1$, il vient que $W_j \in]\frac{1}{4}, \frac{7}{4}[$. On peut en déduire que $W_{j+1} \in]-\frac{7}{4}, \frac{7}{4}[$. Le cas où $s_j = -1$ est parfaitement symétrique. Pour le dernier cas, le fait que s_j soit nul implique que $W_j \in]-\frac{1}{2}, \frac{1}{2}[$ et donc $W_{j+1} \in]-\frac{7}{4}, \frac{7}{4}[$. Nous verrons plus en détail la justification de l'exactitude de l'algorithme dans le cas de la division, dont l'opérateur est plus complexe. On obtient les propriétés suivantes à l'aide du premier invariant :

$$\left|S_j - \frac{X_j + Y_j}{16}\right| < 2^{-j} \quad \text{ou bien} \quad \lim_{j \to +\infty} S_j = \frac{X+Y}{16}$$

Multiplication L'opérateur de multiplication en-ligne calcule à l'itération de rang j les produits partiels qui font intervenir x_j et y_j qu'il n'a pas pu calculer par le passé parce que ces chiffres des opérandes n'étaient pas disponibles. Pour ce faire, il effectue les produits $y_j X_j$ et $x_j Y_{j-1}$, où X_j est constitué des j premiers chiffres de X et Y_{j-1} des $j-1$ premiers chiffres de Y. Le circuit doit donc stocker les chiffres de X et de Y au fur et à mesure qu'ils sont transmis à l'opérateur. Le reste de la récurrence sert à calculer au fur et à mesure les chiffres qui n'évolueront plus dans l'avenir et que l'on peut d'ores et déjà retourner à l'utilisateur.

L'opérateur de multiplication est très différent de l'opérateur d'addition parce que les trois registres X_j, Y_j et W_j doivent être stockés par l'opérateur. La taille de ces registres dépend de la taille des opérandes. J-M. Muller a montré [Mul 94] qu'il est impossible de calculer en-ligne le produit de deux nombres de longueur arbitraire avec un opérateur de taille fixe. Dans la pratique, cela signifie que tous les multiplieurs en-ligne implantés sont limités : par exemples les implantations de A. Skaf [Ska 95] utilisent des registres de 16 chiffres signés et nous détenons probablement le record d'implantation sur FPGA avec des registres de 605 chiffres pour la multiplication. Nous verrons à la fin de ce chapitre que l'unité Puce par sa conception originale peut s'affranchir de cette limite de façon transparente en utilisant de la mémoire. Nous ne sommes plus alors limités par la taille de l'opérateur, mais par la taille de la mémoire.

$$\begin{array}{ll} X_j & = X_{j-1} + x_j 2^{-j} \\ Y_j & = Y_{j-1} + y_j 2^{-j} \\ p_j & = sign_2(2W_{j-1}) \\ W_j & = 2W_{j-1} - p_j + \frac{y_j X_j + x_j Y_{j-1}}{16} \\ P_j & = P_{j-1} + p_j 2^{-j} \end{array}$$

Les invariants de boucle sont :

$$P_j + W_j 2^{-j} = \frac{X_j Y_j}{16} \quad \text{et} \quad W_j \in \left]-\frac{7}{8}, \frac{7}{8}\right[$$

Pour l'initialisation de l'opérateur et les hypothèses sur les opérandes nous utilisons les mêmes valeurs que pour l'addition. Il s'ensuit que

$$\left| P_j - \frac{X_j Y_j}{16} \right| < 2^{-j} \quad \text{ou bien} \quad \lim_{j \to +\infty} P_j = \frac{XY}{16}$$

Exemple de Fonctionnement Nous allons calculer en-ligne le produit $\frac{3}{8} \times \frac{1}{8}$. Nous écrivons les données de la manière suivante. Le résultat est bien contenu dans P_{11} à la fin de l'algorithme, il s'agit de $\frac{3}{64}$, ce qui donne $P_{11} = \frac{3}{64 \times 16}$.

$$X = 0.10\overline{1}0... \quad \text{et} \quad Y = 0.1\overline{11}0...$$

Itération		
1	$X_1 = 0.1$ $Y_1 = 0.1$	$p_1 = sign(0.00) = 0$
	$W_1 = 0 - 0 + \frac{1\times 0.1 + 1\times 0}{10000} = 0.00001$	$P_1 = 0.0$
2	$X_2 = 0.10$ $Y_2 = 0.1\overline{1}$	$p_2 = sign(0.00) = 0$
	$W_2 = 0.0001 - 0 + \frac{\overline{1}\times 0.10 + 0\times 0.1}{10000} = 0.0001\overline{1}$	$P_2 = 0.00$
3	$X_3 = 0.10\overline{1}$ $Y_3 = 0.1\overline{11}$	$p_3 = sign(0.00) = 0$
	$W_3 = 0.001 - 0 + \frac{\overline{1}\times 0.10\overline{1} + \overline{1}\times 0.10}{10000} = 0.0000011$	$P_3 = 0.000$
4		$p_4 = sign(0.00) = 0$
	$W_4 = 0.000011 - 0 = 0.000011$	$P_4 = 0.0000$
5		$p_5 = sign(0.00) = 0$
	$W_5 = 0.00011 - 0 = 0.00011$	$P_5 = 0.00000$
6		$p_6 = sign(0.00) = 0$
	$W_6 = 0.0011 - 0 = 0.0011$	$P_6 = 0.000000$
7		$p_7 = sign(0.01) = 0$
	$W_7 = 0.011 - 0 = 0.011$	$P_7 = 0.0000000$
8		$p_8 = sign(0.11) = 1$
	$W_8 = 0.11 - 1 = 0.\overline{1}1$	$P_8 = 0.00000001$
9		$p_9 = sign(0.\overline{1}1) = \overline{1}$
	$W_9 = \overline{1}.1 - \overline{1} = 0.1$	$P_9 = 0.00000001\overline{1}$
10		$p_{10} = sign(1.00) = \overline{1}$
	$W_{10} = 1 - 1 = 0$	$P_{10} = 0.00000001\overline{1}1$

Division L'algorithme naturel de division tel qu'il est enseigné à l'école primaire s'écrit, une fois adapté à la base 2, $W_j = 2W_{j-1} - q_{j-1}D$. En effet, si W_0 est le dividende et D le diviseur, le chiffre $j-1$ du quotient est q_{j-1}. La version en-ligne de cet algorithme est identique. Il faut toutefois à l'itération j compenser les erreurs commises parce que l'opérateur ne connaît pas l'ensemble des chiffres du dividende et du diviseur dès le début du calcul. On corrige l'erreur en jouant sur les diverses écritures des nombres.

Afin de garantir un bon fonctionnement, le diviseur subit une pseudo-normalisation qui consiste à le multiplier par une puissance de 2 adéquate pour garantir que $1 - \frac{1}{8} \leq |D| < 2$. Au besoin, on modifiera la valeur de X pour garantir que $|Q| \leq \frac{1}{16}$.

$$\begin{array}{ll} X_j & = X_{j-1} + x_j 2^{-j} \\ D_j & = D_{j-1} + d_j 2^{-j} \\ q_j & = sign_2(2W_{j-1}) \\ W_j & = 2W_{j-1} - q_j D_j - d_j Q_{j-1} + \frac{x_j}{32} \\ Q_j & = Q_{j-1} + q_j 2^{-j} \end{array}$$

Les invariants de boucle sont :

$$D_j Q_j + W_j \times 2^{-j} = \frac{X_j}{32} \quad \text{et} \quad W_j \in \left]-|D_j| + \frac{1}{8}, |D_j| - \frac{1}{8}\right[$$

On en déduit alors que :

$$\left|Q_j - \frac{X_j}{32D_j}\right| < \frac{16}{5} 2^{-j} \quad \text{ou bien} \quad \lim_{j \to +\infty} Q_j = \frac{X}{32D}$$

Correction de l'Algorithme Nous allons démontrer l'ensemble des propriétés qui sont données pour la division. Sans nuire à la généralité du problème on pose $D > 0$. Les deux invariants sont vrais pour $j = 0$. Nous allons supposer qu'ils sont vrais pour un nombre $j - 1$ quelconque.

Le premier invariant s'écrit à l'ordre j :

$$D_j Q_j + W_j \times 2^{-j} = \frac{X_j}{32}$$

Ceci est équivalent à la proposition suivante :

$$(D_{j-1} + d_j 2^{-j})\,(Q_{j-1} + q_j 2^{-j}) + \left(2W_{j-1} - q_j D_j - d_j Q_{j-1} + \frac{x_j}{32}\right) 2^{-j} = \frac{X_{j-1} + x_j 2^{-j}}{32}$$

En développant, on vérifie que la proposition à l'ordre j est équivalente à la proposition à l'ordre $j - 1$. Le premier invariant est donc démontré par récurrence.

$$\begin{array}{c} D_{j-1}Q_{j-1} + D_{j-1}q_j 2^{-j} + d_j 2^{-j} Q_{j-1} + d_j 2^{-j} q_j 2^{-j} + 2W_{j-1}2^{-j} - q_j D_j 2^{-j} - d_j Q_{j-1} 2^{-j} + \frac{x_j}{32} 2^{-j} \\ = \frac{X_{j-1}}{32} + \frac{x_j}{32} 2^{-j} \end{array}$$

Le second invariant à l'ordre $j - 1$ est équivalent à :

$$W_{j-1} \in \left]-D_{j-1} + \frac{1}{8}, D_{j-1} - \frac{1}{8}\right[$$

Nous allons mener une étude cas par cas selon le signe de q_j.

– Dans le cas où $q_j = 1$, on déduit que $Tronc_2(2W_{j-1}) \geq \frac{1}{2}$ ce qui implique que :

$$\frac{1}{4} < 2W_{j-1} < 2D_{j-1} - \frac{1}{4}$$

Il vient alors que :

$$\frac{1}{4} - D_j < 2W_{j-1} - q_j D_j < D_j - 2d_j 2^{-j} - \frac{1}{4}$$

Et on obtient pour W_j :

$$\frac{1}{4} - D_j + d_j Q_{j-1} + \frac{x_j}{32} < W_j < -\frac{1}{4} + D_j - 2d_j 2^{-j} + d_j Q_{j-1} + \frac{x_j}{32}$$

D'où pour $j > 5$:

$$\begin{array}{rl} \frac{1}{4} - D_j + d_j Q_{j-1} + \frac{x_j}{32} & > \frac{1}{4} - D_j - \frac{1}{16} - \frac{1}{32} \\ & > \frac{1}{8} - D_j \\ -\frac{1}{4} + D_j - 2d_j 2^{-j} + d_j Q_{j-1} + \frac{x_j}{32} & < -\frac{1}{4} + D_j - \frac{1}{32} + \frac{1}{16} + \frac{1}{32} \\ & < D_j - \frac{1}{8} \end{array}$$

D'après les hypothèses, $|D| \geq 1 - \frac{1}{16}$ et $|Q| < \frac{1}{16}$, ce qui implique que $|X| < \frac{1}{8}$ et donc $W_j = 0$ pour $j \leq 3$. Pour $j = 4$, on trouve trivialement que $W_4 = \frac{x_4}{32}$ ce qui est amplement suffisant et garantit que $q_5 = 0$ et donc que $W_3 \leq \frac{1}{8}$.

– Dans le cas où $q_j = 0$, on déduit que $Tronc_2(2W_{j-1}) \in [-\frac{1}{2}, \frac{1}{2}[$ ce qui implique que :

$$-\frac{1}{2} < 2W_{j-1} < \frac{1}{2}$$

Et on obtient pour W_j :

$$-\frac{1}{2} + d_j Q_{j-1} + \frac{x_j}{32} < W_j < \frac{1}{2} + d_j Q_{j-1} + \frac{x_j}{32}$$

D'où pour $j > 4$:

$$\begin{array}{rl} -\frac{1}{2} + d_j Q_{j-1} + \frac{x_j}{32} & > -D_j + \frac{3}{8} - \frac{1}{16} - \frac{1}{32} \\ & > \frac{1}{8} - D_j \\ \frac{1}{2} + d_j Q_{j-1} + \frac{x_j}{32} & < D_j - \frac{3}{8} + \frac{1}{16} + \frac{1}{32} \\ & < D_j - \frac{1}{8} \end{array}$$

Dans les autres cas, on sait que $W_j = 0$. Ces cycles où l'algorithme semble *tourner à vide* sont nécessaires parce que l'algorithme ne peut fonctionner que si l'opérateur connaît déjà quelques chiffres du diviseur.

Racine Carrée L'opérateur d'extraction de racine carrée en-ligne est très proche de l'opérateur de division en-ligne. Les algorithmes usuels de division et de racine carrée sont aussi voisins. Un additionneur supplémentaire est utilisé pour accumuler q_j^2 à la bonne position dans l'accumulateur W_j. Les même hypothèses sont nécessaires pour cet opérateur.

$$\begin{array}{ll} X_j & = X_{j-1} + x_j \times 2^{-j} \\ q_j & = sign_2(2 \times W_{j-1}) \\ W_j & = 2W_{j-1} - 2q_j Q_{j-1} + q_j^2 \times 2^{-j} + \frac{x_j}{32} \\ Q_j & = Q_{j-1} + q_j \times 2^{-j} \end{array}$$

Les invariants de boucle sont :

$$Q_j^2 + W_j \times 2^{-j} = \frac{X_j}{32} \quad \text{et} \quad W_j \in \left]-Q + \frac{1}{8}, Q - \frac{1}{8}\right[$$

On en déduit alors que :

$$\left|Q_j - \sqrt{Xj}\right| < 2^{-\lfloor \frac{j}{2} \rfloor} \quad \text{ou bien} \quad \lim_{j \to +\infty} Q_j = \sqrt{X}$$

2.2 Accélérateur Matériel PeRLe

Les architectures modernes ont souvent recours à un jeu d'instructions réduit (technologie RISC), pour allonger la longueur de leurs *pipelines* de données et d'instructions et atteindre ainsi des vitesses d'horloge et des puissances de calcul toujours plus élevées. Certains problèmes sont néanmoins mieux traités par une unité spécialisée que par les microprocesseurs disponibles actuellement : par exemple, les algorithmes de traitement du signal et de traitement d'image qui sont de très gros consommateurs de calculs élémentaires répétitifs.

Ces algorithmes n'utilisent qu'un nombre réduit d'opérations relativement simples du jeu d'instructions offert par un microprocesseur généraliste ; ils n'ont aucunement besoin de toutes les capacités disponibles sur une machine universelle complète. Un circuit dédié qui effectue uniquement les tâches simples et répétitives d'un algorithme de traitement du signal sera beaucoup plus petit qu'un microprocesseur. On pourra par la suite intégrer plusieurs copies de ce même circuit sur une seule puce ; la régularité des opérations implantées permet alors d'allonger encore les *pipelines* et en conséquence d'augmenter la vitesse d'horloge. Le calcul efficace des fonctions de l'arithmétique flottante passe aussi par la construction sur le circuit d'opérateurs dédiés comme c'est le cas dans les unités flottantes qui sont intégrées dans les microprocesseurs. Les opérations de normalisation et particulièrement l'arrondi avec bit de garde spécifié dans la norme IEEE ne peuvent pas être effectuées sur l'unité arithmétique et logique de calcul entier du microprocesseur. Pour l'arithmétique en-ligne, les opérations utilisées ne sont pas bien adaptées aux processeurs usuels sauf dans des environnements très spécifiques [Maz 93].

Construire un circuit dédié est coûteux. La phase de conception du circuit met en jeu des algorithmes longs de placement et de routage qui sont connus pour être à la limite de la complexité accessible aux ordinateurs. Pour obtenir un circuit efficace en regard des technologies utilisées il faut maintenir une interaction permanente entre d'une part les logiciels de placement et de routage et d'autre part le concepteur. À tous les niveaux de la conception, la description symbolique du circuit doit être comparée avec une modélisation de niveau supérieur pour garantir la validité du circuit. Parce que le coût financier de la fonte d'un circuit prototype est très élevé, il est primordial que les phases de tests soient effectuées avec le plus grand soin, et que le nombre de prototypes nécessaires, avant d'obtenir un circuit définitif correct, soit minimum.

Ainsi, les phases de description et de validation d'un circuit dédié peuvent prendre de deux à six mois suivant la complexité du circuit, la taille de l'équipe de concepteurs et leur savoir-faire. Les technologies reconfigurables telles que les prédiffusés actifs allègent les contraintes des concepteurs. Les tests de validation du circuit peuvent être menés directement sur le circuit, en grandeur réelle et sans recours à un modèle d'exécution interne. C'est ainsi que l'on vérifie plus aisément le bon fonctionnement du circuit. De plus, si une erreur apparaît après que le circuit ait passé avec succès les premiers tests de validation, il est toujours possible de modifier la configuration pour un coût ajouté très faible.

Les circuits programmables offrent au concepteur une souplesse d'utilisation comparable à un élément logiciel. En terme d'efficacité, le temps de commutation des prédiffusés actifs est plus long que celui d'un VLSI d'un facteur compris entre 10 et 20, selon les usages. Pour obtenir quand même une puissance de calcul intéressante sur prédiffusés actifs, nous avons massivement recours au *pipeline* et au parallélisme à grain fin. Nous obtenons couramment sur les projets que nous avons réalisés des vitesses d'horloge allant de 2 MHz pour des circuits de validation d'algorithmes à des vitesses de 10 MHz à 33 MHz pour les circuits définitifs.

La carte PeRLe, construite par le groupe de J. Vuillemin au laboratoire PRL de DEC à Paris s'intègre dans l'environnement d'un ordinateur. Avec en son cœur une matrice de calcul de 16

circuits programmables, la carte PeRLe peut prendre le relais de l'unité centrale de l'ordinateur pour des applications peu adaptées à l'architecture du microprocesseur. À chaque type de problème correspond une configuration à charger sur l'accélérateur matériel. La carte PeRLe agit dès lors comme un coprocesseur spécialisé dédié à chaque projet. On peut configurer ce coprocesseur pour décharger l'unité centrale des tâches simples et répétitives ou des opérations mal adaptées à l'arithmétique entière de 16 ou 32 bits.

2.2.1 Architecture Générale

La carte PeRLe-1 de *Digital Equipment* est, selon le terme de J. Vuillemin, un exemple de mémoire active programmable ou *Programmable Active Memory* (PAM) [BRV 89]. Elle est basée sur la technologie des prédiffusés actifs. La configuration des composants de la carte est chargée depuis la machine hôte. Une reconfiguration complète peut intervenir en quelques dixièmes de seconde grâce à l'interface rapide TURBOchannel. Les données qui sont échangées entre la carte et la machine hôte exploitent le même lien rapide. L'ensemble de la conception de la carte PeRLe et des mécanismes de communication avec la machine en font un *accélérateur matériel* capable de simuler efficacement de nombreux circuits.

La carte PeRLe regroupe 23 circuits intégrés reconfigurables Xilinx XC 3090. La carte d'interface elle-même qui permet la connexion entre la carte PeRLe et la station de travail contient un circuit Xilinx. L'ensemble du comportement de la carte peut être modifié ou reprogrammé si un problème devait apparaître. Nous nous intéresserons néanmoins plus en détail aux seize circuits positionnés selon une grille à deux dimensions que l'on appelle la matrice de calcul. Sept circuits reconfigurables supplémentaires ont pour charge d'acheminer les données et de gérer les fonctions de contrôle et de communication avec l'hôte accessible à l'utilisateur.

Une vue détaillée de la carte PeRLe avec les liens de communication est présentée Figure 2.7. La matrice de calcul est entourée de quatre bancs de mémoire statique rapide de 256 Mmots de 32 bits soit un total de 4 Mo. La vitesse des mémoires implantées permet de réaliser des transactions avec un cycle d'horloge de 50 ns. Les circuits périphériques ont en charge l'acheminement des signaux entre la matrice de calcul, l'hôte et les mémoires. Il se décomposent comme suit :

- Quatre commutateurs (*switch*) sont placés autour de la matrice selon les directions cardinales (N, E, S, W). Chaque commutateur possède une liaison de 32 bits vers le banc mémoire associé, une liaison vers son contrôleur et une liaison de 64 bits vers la matrice. Le bus de 64 bits connectant la matrice avec le commutateur est distribué selon les rangées ou les colonnes de la matrice. Pour le commutateur nord, chacune des 4 colonnes de la matrice a accès à 16 bits consécutifs du bus. Ces signaux sont partagés par tous les circuits d'une même colonne de la matrice de calcul. Il sont connectés à 16 fils du circuit sur la face correspondant à l'orientation du commutateur. Les commutateurs possèdent d'autres accès, plus restreints, à la matrice de calcul. Nous les verrons par la suite. Le lien FIFO et les commutateurs peuvent également prendre en charge une préparation simple des données avant de les envoyer à la matrice de calcul.

- Les données sont fournies aux commutateurs par le lien FIFO. Les contrôleurs commandent les signaux de contrôle externes de la carte qui gèrent, entre autres, les échanges de données avec la machine hôte. Le contrôleur nord-est regroupe les commutateurs nord et est, le contrôleur sud-ouest les commutateurs sud et ouest. Les signaux de contrôle de la mémoire sont aussi générés par les contrôleurs correspondants. Les deux contrôleurs sont connectés directement par quelques signaux pour échanger des données de synchronisation.

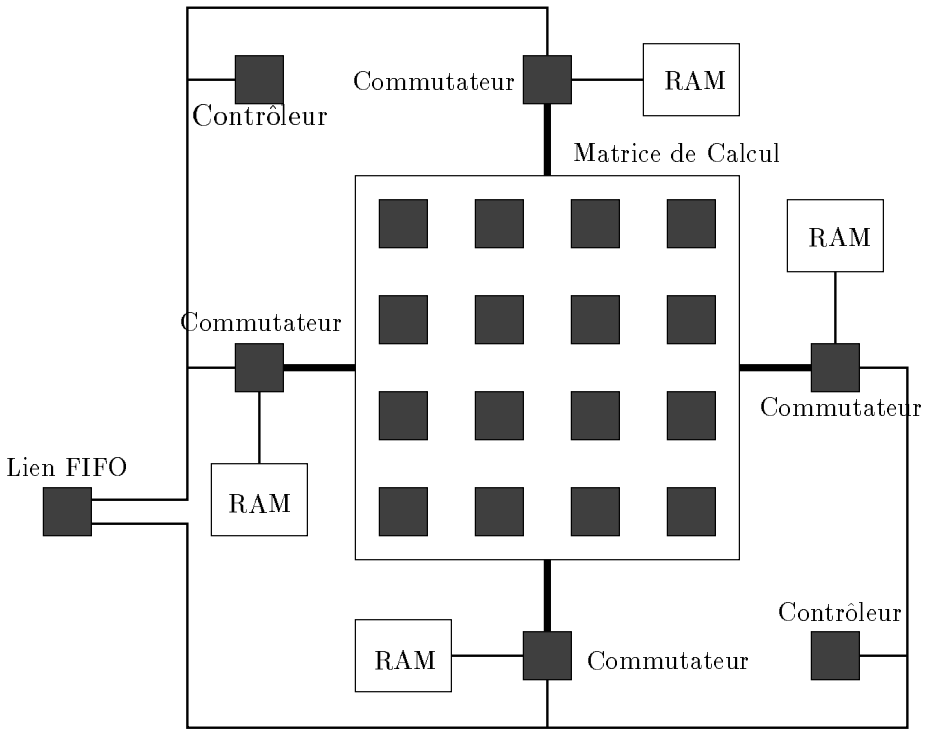


FIG. 2.7 - *Carte PeRLe 1 — Vue d'ensemble*

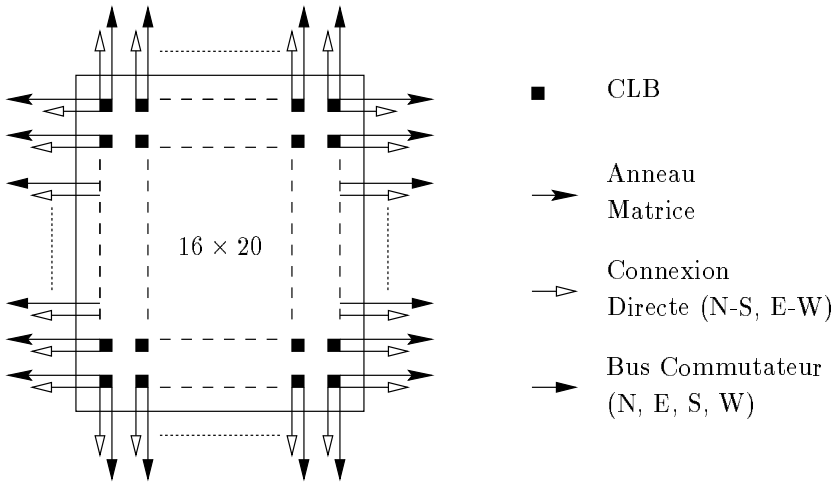


FIG. 2.8 - *Schéma d'Interconnexion d'un Circuit*

- Une file d'attente *First In—First Out* (FIFO) assure la liaison entre la carte et le contrôleur de bus. Le contrôleur de bus assure les communications de bas niveau avec la machine hôte. Bien qu'il soit possible de le reprogrammer nous n'envisagerons pas ce cas dans notre étude. Le FIFO possède une liaison 32 bits avec la machine en entrée et une en sortie. Ces deux liaisons sont reprises à destination des commutateurs.

Tous les circuits de la carte sont connectés par divers mécanismes. Cela permet une distribution rapide des données sur la carte. On peut ainsi mettre en œuvre des circuits aux comportements très différents. Le réseau d'interconnexion est le plus dense au sein de la matrice de calcul où chaque circuit appartient à sept réseaux différents.

Le réseau le plus rapide connecte uniquement les circuits de la matrice. Il s'agit d'une liaison en grille à deux dimensions de 64 fils de large dans chaque direction. Chaque circuit est connecté à ses quatre plus proches voisins par un bus bi-directionnel de 16 bits. Ce réseau de communication reprend le réseau interne au circuit de communication vers le plus proche voisin. Il permet une communication de proximité. Le réseau n'est pas configuré selon un tore, mais d'après une grille. Les 256 fils qui sont laissés libres sont reliés à des connecteurs sur la carte. Ces connecteurs peuvent être utilisés pour obtenir des données en provenance d'un dispositif annexe ou bien être utilisés en sortie. Cela permet de connecter un convertisseur analogique numérique en entrée, un convertisseur numérique analogique en sortie, un échantilloneur ou un amplificateur audio *etc*...

Nous venons de voir les deux réseaux de connexion directe qui établissent le lien entre les lignes et les colonnes de la matrice. Les quatre bus des commutateurs relient les lignes et les colonnes de la matrice avec les commutateurs. Un dernier bus de dix bits seulement connecte l'ensemble des circuits de la matrice et les commutateurs. L'ensemble constitue les sept réseaux d'interconnexion accessibles sur la carte (voir Figure 2.8).

Le temps de propagation des signaux dans les divers circuits varie : la matrice de connexion directe offre les connections les plus rapides avec un temps de commutation de 24 ns ; les bus des commutateurs sont utilisés à 28 ns ; le délai pour le bus global est plus important, 43 ns. Au niveau de la carte, les circuits performants sont ceux qui utilisent au mieux les communications entre les circuits en faisant appel à des communications vers les plus proches voisins.

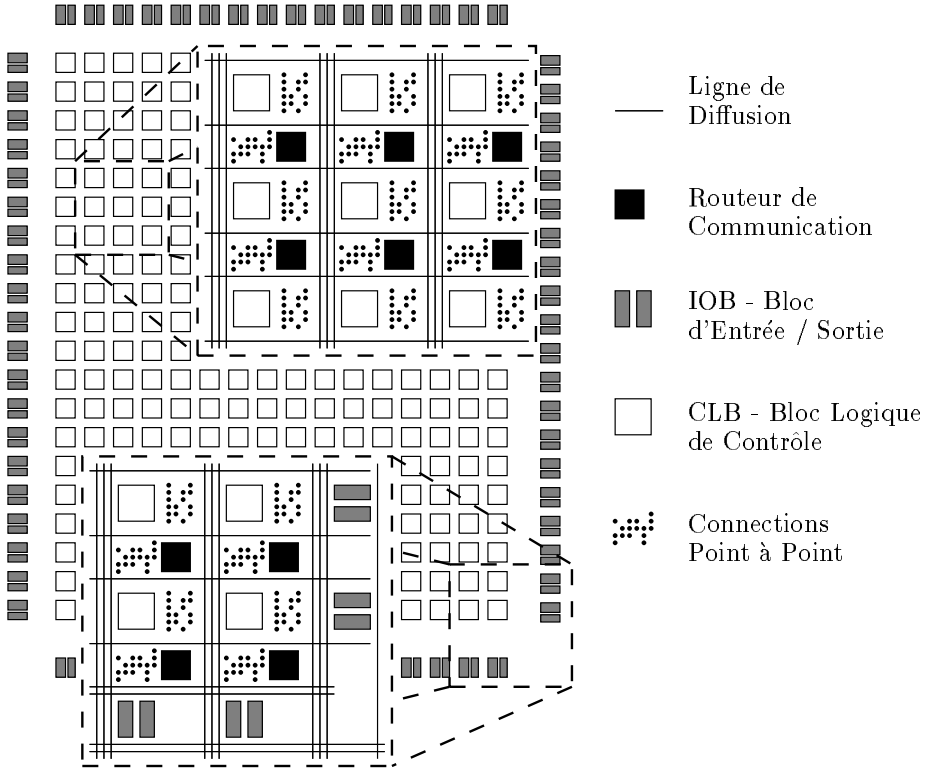


FIG. 2.9 - *Architecture Interne du Xilinx XC 3090*

Les transferts de données entre la carte et la machine hôte se font à travers un bus de 32 bits de large. Quand le lien est configuré en accès direct à la mémoire (DMA), les échanges sont effectués à la vitesse de 100 Mo/s. Cela correspond à une carte produisant des données en continu à une vitesse d'horloge de 25 MHz. Les vitesses usuelles de travail pour un circuit vont de quelques MHz à 25 MHz. On peut néanmoins obtenir pour des circuits particulièrement optimisés une vitesse de calcul plus élevée. Au delà de 40 MHz, toutes les communications doivent être gérées avec un soin particulier.

2.2.2 Prédiffusé Actif

Le circuit intégré Xilinx XC 3090 [Xil 94] est construit autour d'un noyau de 20×16 cellules logiques appelées *Control Logic Blocks* (CLB) (voir Figure 2.9). Il s'agit d'un circuit intégré de la classe des *Logic Cell Arrays* (LCA). Pour chaque cellule, le circuit possède des capacités de routage à faible distance et des capacités de diffusion rapide. Le noyau est entouré d'un anneau de 144 ports d'entrées/sorties. On appelle ces ports les *Input-Output Block* (IOB).

Chaque cellule logique est capable d'implanter deux fonctions dont le nombre d'arguments est limité à sept pour les deux fonctions (voir Figure 2.10). On peut, au choix, décider d'implanter deux fonctions de quatre variables, dont seulement une variable est en commun, ou deux fonctions des mêmes cinq variables et, même, une fonction de sept variables. La valeur de la fonction est stockée pour chaque configuration des entrées dans une table. Le calcul est alors réalisé par la lecture de cette table. Chaque cellule contient également deux registres binaires qui sont situés sur le chemin

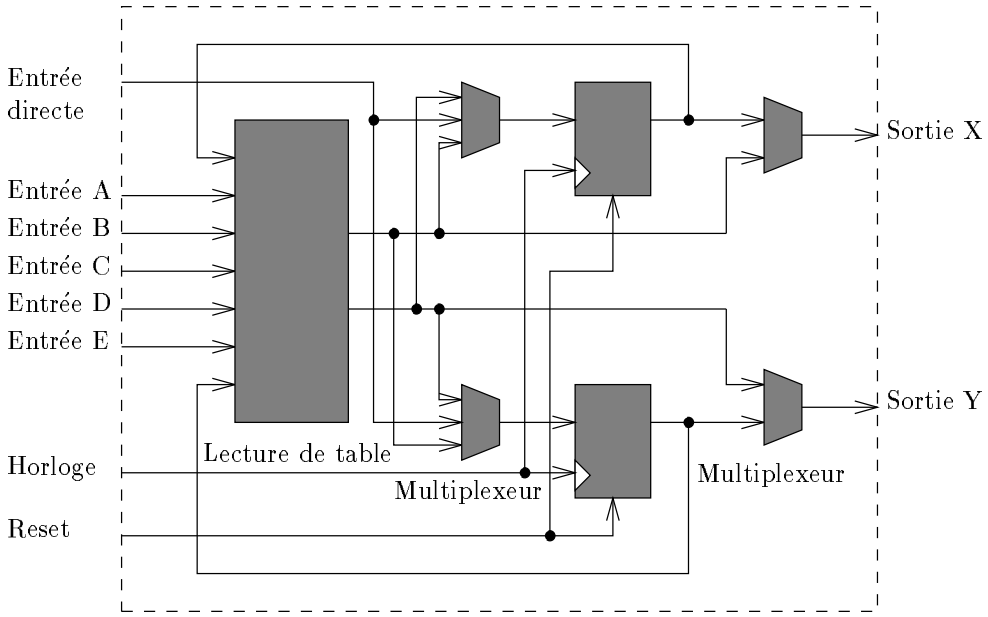


FIG. 2.10 - *Cellule Logique* — Control Logic Block *(CLB)*

de données afin d'enregistrer les résultats des deux fonctions. Les deux registres peuvent aussi être utilisés comme entrées des fonctions. On peut donc implanter des automates cellulaires simples. Les deux registres sont configurés indépendamment l'un de l'autre, mais si on enregistre une donnée directement sur les registres, un seul fil est disponible en entrée du CLB. De ce fait si les deux registres lisent cette donnée en même temps, ils stockeront la même valeur.

Les signaux sont transmis aux CLB à l'aide d'un réseau de routeurs et de connections programmables. Les circuits qui favorisent les communications avec les voisins directs utilisent principalement les connections point à point programmables. On considère souvent en première approximation que le temps de transmission de l'information est nul à travers une connexion programmable. Pour ces raisons, un circuit favorisant les communications vers les proches voisins bénéficiera d'une pénalité de communication beaucoup plus faible, le temps de commutation d'un routeur étant de l'ordre de quelques milli-secondes. Une nappe de cinq fils circule à droite et en dessous de tous les CLB et connecte les plots des CLB et les plots des routeurs. On peut transmettre une information à une cellule située à quelque cellules de distance de l'émettrice en faisant transiter le signal par ces nappes et par des routeurs de communication successifs. Néanmoins, les routeurs ne peuvent réaliser plus de vingt connections et sont rapidement saturés dans le cas d'un circuit à fort taux d'intégration.

Quand une donnée doit être transmise à un grand nombre de cellules, on peut utiliser les lignes de diffusion. Ces lignes sont au nombre de deux pour chaque colonne et de deux pour chaque ligne d'un circuit. Si une colonne n'utilise pas le signal d'horloge, une troisième ligne est disponible sur cette colonne. Les lignes et les colonnes peuvent être coupées en leur milieu si la configuration du circuit le demande. On ne peut malheureusement pas connecter directement une ligne verticale et une ligne horizontale, ce qui serait utile quand on veut diffuser un signal à tous les CLB d'un circuit. Dans ce cas, il faut faire transiter le signal par un CLB. Nous avons remarqué que le bon usage des lignes de diffusion est essentiel pour la conception d'un circuit rapide et à haute intégration. Les

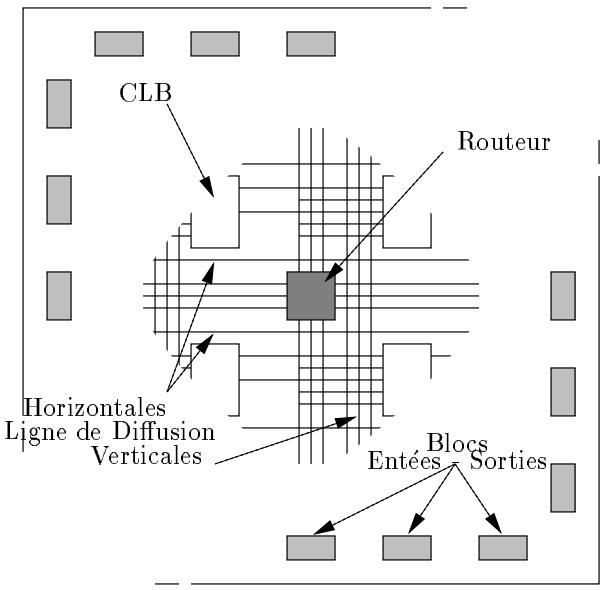


FIG. 2.11 - *Signaux à l'Intérieur d'un Circuit*

lignes de diffusion horizontales peuvent également être utilisées comme des bus à trois états ce qui permet aussi d'implanter l'opération ou câblée. L'ensemble des réseaux est repris Figure 2.11.

Concevoir un circuit sur prédiffusé actif est une tâche très différente de celle de concevoir un circuit intégré. Pour être efficace, il faut adapter le circuit à la logique de bloc des prédiffusés actifs. Cela pourra nécessiter quelques changements dans l'algorithme implanté. Dans le cas de l'arithmétique des ordinateurs, nous avons considéré l'utilisation de bases de numération autres que la base 2. Le nombre limité de ressources et leur distribution uniforme peuvent entraîner d'autres problèmes de répartition. Ces problèmes sont amplifiés par la spécialisation de certaines connexions. Ainsi les lignes de diffusion ne peuvent être connectées qu'à un nombre restreint de plots d'entrée d'un CLB. Ces contraintes font que le placement et le routage d'un circuit sur prédiffusé actif posent autant de problèmes que l'adaptation à la structure par bloc.

2.2.3 Programmer la Carte PeRLe

La bibliothèque de programmation PeRLe1DC conçue au PRL [Tou 92] permet de générer l'image mémoire de la carte PeRLe correspondant à la configuration que l'on définit. La carte regroupe un total de plus de 23 milliers de CLB chacun nécessitant sept bits de configuration pour les seules opérations logiques. Si on ajoute à cela les informations de routage et les plots d'entrée sortie, cela fait un total de 100 Ko où chaque bit doit avoir exactement la valeur qui lui est assignée. La bibliothèque est écrite en C++ et elle utilise les nombreuses constructions des langages orientés objets :

Surcharge des Opérateurs Les opérations logiques qui conditionnent le fonctionnement du circuit sont spécifiées à l'aide des équations booléennes sur les signaux. L'utilisateur peut faire directement référence à des variables intermédiaires, à des registres pour les circuits synchrones, à un ou plusieurs signaux d'horloge. Les opérateurs booléens usuels, les multiplexeurs et les

constantes telles que la valeur `vrai` ou `faux` sont accessibles à l'aide des opérateurs usuels du langage.

Héritage et Généricité Les structures de données avancées telles que les vecteurs de signaux dont la largeur peut être statique ou dynamique sont implantées directement par le biais des classes C++ ; le langage reconnaît un type propre aux équations *etc*... Nous avons exploité ces constructions dans la définition de la cellule Nacel et pour l'utiliser dans les opérateurs de multiplication, de division et d'extraction de racine carrée.

Par défaut, le placement et le routage des circuits sont effectués automatiquement par les outils Xilinx. La bibliothèque permet néanmoins d'utiliser directement toute la flexibilité des outils de placement et de routage. On peut placer n'importe quelle partie du circuit et laisser les outils Xilinx placer les autres éléments du circuit. Le langage permet de placer des cellules de relativement à d'autres cellules. Ainsi nous verrons comment on spécifie la forme d'une cellule PPM générique sans la placer. Néanmoins, les outils Xilinx ne prennent en compte que des contraintes absolues de placement. Les contraintes relatives sont gérées par la bibliothèque PeRLe1DC. De ce fait, on ne peut maintenir de contraintes de forme sur un groupe de cellule sans le placer en fixant la position d'une de ses cellules.

On peut assigner une priorité à un signal afin que les signaux critiques soient pris en compte en premier par les outils de routage. Avec la version actuelle de la bibliothèque, on ne peut pas assigner un fil à un signal. On peut toutefois positionner un signal sur un port d'entrée ou de sortie, ce qui a pour effet de restreindre les possibilités de routage, et dans bien des cas de fixer le signal à l'endroit que le concepteur a choisi. Les lignes de diffusion horizontales peuvent être utilisées comme des bus à trois états, de ce fait, on peut y accéder directement à partir du langage de conception.

Les circuits conçus à l'aide de la bibliothèque sont simulés à plusieurs niveaux de la conception par les outils de PRL (niveau symbolique) et de Xilinx (niveau électrique). Les fichiers générés par la bibliothèque sont compatibles avec tous les outils Xilinx, cela permet de les utiliser dans les phases suivantes de la conception.

2.3 Bibliothèque d'Opérateurs sur PeRLe-1

2.3.1 Addition

Le schéma logique de l'additionneur en-ligne a été présenté plus tôt dans ce chapitre. Afin d'implanter l'additionneur en-ligne sur PAM, nous avons utilisé seulement deux cellules logiques (voir Figure 2.12). Des quatre registres disponibles sur les deux cellules logiques, seuls trois sont utilisés. Pour exemple, voici le code que nous utilisons pour générer une cellule PPM en utilisant les fonctions de base de la bibliothèque PeRLe1DC. La dernière instruction impose au compilateur de générer les signaux u et v sur le même CLB. On peut ainsi définir la *forme* de l'opérateur sans être obliger de le placer définitivement, ce qui permet de l'utiliser plusieurs fois.

```
void PlusPlusMoins (Bool &a, Bool &b, Bool &c, Bool &u, Bool &v) {

  u  = (a & b) | (b & ~c) | (a & ~c);
  v  = a ^ b ^ c;
  v <= u;
}
```

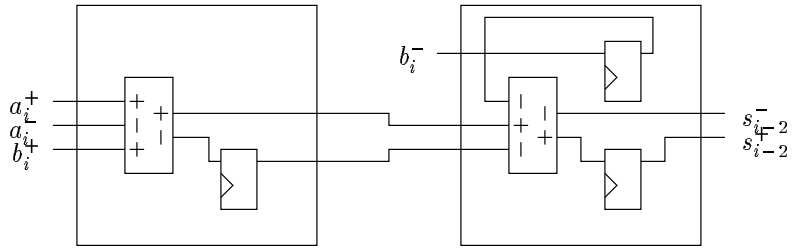


FIG. 2.12 - *Cellules FPGA de l'Additionneur En-ligne*

L'additionneur en-ligne est très facile à définir à l'aide de la fonction `PlusPlusMoins` et des primitives du langages. La structure de données `Digit` que nous avons défini pour un chiffre en BS s'intègre parfaitement à la bibliothèque de conception et nous pouvons ainsi définir un bus de chiffres binaires signés.

```
void OnLineAdder (Digit &a, Digit &b, Digit &out, Bool &Reset,
                  WireVector <Digit, 3> &temp)
{
  PlusPlusMoins   (a[0], b[0], a[1], temp[0][0], temp[0][1]);
  ResetRegister   (temp[0][1], temp[1][0], Reset);
  temp[1][0]   <= temp[0][1];

  PlusPlusMoins   (temp[1][0], temp[1][1], temp[0][0], temp[2][1], temp[2][0]);
  ResetRegister   (b[1], temp[1][1], Reset);
  ResetRegister   (temp[2][0], out[0], Reset);
  out[1]        = temp[2][1];

  temp[1][1]   <= temp[2][1];
  temp[2][0]   <= out[0];
  temp[2][1]   <= out[1];
}
```

2.3.2 Noyau Arithmétique de Calcul En-Ligne

Les opérateurs de multiplication, de division et d'extraction de racine carrée partagent la même organisation générale : le résultat intermédiaire est décalé vers la gauche puis ajouté à deux autres termes obtenus par la multiplication d'un nombre et d'un chiffre (voir Figure 2.13). Le circuit commun que nous appelons le Noyau Arithmétique de Calcul En-Ligne (Nacel) calcule la somme des trois opérandes puis une éventuelle transformation locale, due au terme c_j, sur les premiers chiffres de $W^{(j)}$.

$$W_j = 2W_{j-1} + a'_j A_j + b'_j B_{j-1} + \frac{c_j}{2^l}$$

Nous avons implanté dans la cellule Nacel deux exemplaires de l'additionneur présenté Figure 2.3 et la logique nécessaire pour les produits $a'_j \times A_j$ et $b'_j \times B_{j-1}$. La transformation locale n'agit que sur quelques chiffres significatifs ; on l'obtient en utilisant peu de cellules logiques supplémentaires. Les mots A_j et B_j sont construits comme suit pour la plupart des opérateurs : le circuit place la

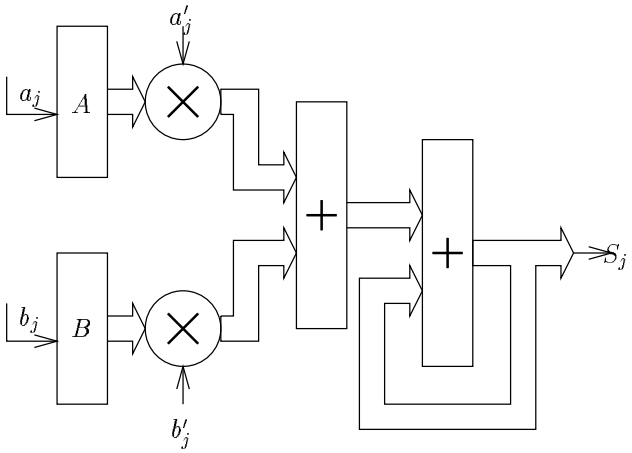


FIG. 2.13 - *Architecture Générale de la Cellule Nacel*

valeur de a^j à la position $j+d$ dans A_j. La constante d sert à décaler de 2^d la valeur de A_j selon l'algorithme de calcul :

$$A_j = A_{j-1} + a_j 2^{-(j+d)}$$

Le circuit compte l'indice j de l'itération en cours avec un curseur: chaque segment k de la cellule Nacel stocke un bit St_k. À l'itération j, si St_k est actif, cela signifie que $k = j$, et donc le chiffre $j+d$ de A_j est positionné suivant la valeur de a_j.

On vérifie que les valeurs suivantes permettent d'évaluer la multiplication, la division et l'extraction de racine carrée à l'aide de la cellule Nacel.

Multiplication Il s'agit de l'opérateur le plus facile à réaliser car la cellule Nacel a été conçue en fonction du multiplieur.

$$\begin{array}{lll} a'_j = y_j & a_j = x_j & A_j = \frac{X_j}{16} \\ b'_j = x_j & b_j = y_j & B_j = \frac{Y_j}{16} \\ & c_j = -p_j & l = 0 \end{array}$$

Division La valeur de l est pénalisant car il faut calculer la transformation locale sur cinq chiffres de plus que dans le cas de la multiplication. L'espace de 16 CLB pour la transformation locale et l'extraction de signe est toujours largement suffisant.

$$\begin{array}{lll} a'_j = -q_j & a_j = d_j & A_j = D_j \\ b'_j = -d_j & b_j = q_j & B_j = Q_j \\ & c_j = x_j & l = 5 \end{array}$$

Extraction de Racine Carrée L'algorithme ne stocke qu'un opérande, l'autre terme de la somme est utilisé pour accumuler la valeur $q_j 2^{-j}$. Le registre de la cellule Nacel $A^{(j)}$ est alors défini comme suit, sans qu'aucun registre ne soit nécessaire. Il est probable que l'on pourrait implanter dans un espace un peu plus faible l'opérateur d'extraction de racine carrée en se

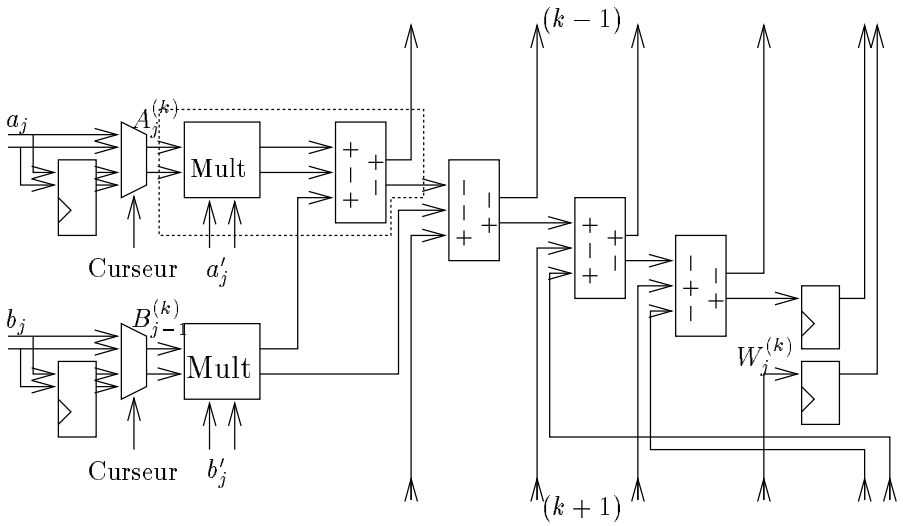


FIG. 2.14 - *Plan d'un Segment au Niveau des Portes Logiques*

passant de la cellule Nacel, mais les gains de place ne dépasseraient pas 10% car, même si les registres ne sont pas utilisés, il faut implanter les fonctions de sélection.

$$\begin{array}{lll} a'_j = q_j & a_j = q_j & A_j = q_j 2^{-j} \\ b'_j = -q_j & b_j = q_j & B_j = 2Q_j \\ & c_j = x_j & l = 5 \end{array}$$

Chaque segment de l'opérateur travaille uniquement sur les chiffres d'un même rang. Il prend soin de récupérer les retenues entrantes et de générer les retenues sortantes pour les tranches suivantes de l'opérateur. La longueur l des registres est fixée par le nombre n de chiffres attendus du résultat. La définition de la valeur minimale de $l = \lceil \frac{n}{2} \rceil$ est détaillée par exemple dans [Kla 93]. Quand n est fixé, il suffit de répéter l fois le segment pour obtenir l'opérateur recherché. En utilisant au mieux les possibilités des cellules logiques de Xilinx, nous avons implanté un segment en huit cellules.

La Figure 2.14 présente le détail du circuit obtenu. Les produits partiels chiffre à chiffre $a'_j \times A_j^{(k)}$, où $A_j^{(k)}$ est le chiffre d'indice k de l'écriture de A_j, et $b'_j \times B_{j-1}^{(k)}$ sont calculés chacun sur une cellule logique. Un additionneur BS a besoin de deux cellules PPM et, comme il faut implanter deux additionneurs, cela fait quatre cellules logiques de plus.

La cellule Nacel est organisée autour d'une classe en C++ appelée Nacel. Chacune des cellules logiques des circuits FPGA est définie séparément dans une fonction individuelle pour rapprocher la conception de l'analyse logique qui vient d'être faite. Nous présentons ici un exemple du programme relativement simple chargé de la multiplication des deux chiffres BS b'_j et $B_{j-1}^{(k)}$.

```
void Nacel :: Cell_AddB (int i, int col, int line) {

  AddB [i][0]      =      mux (Cntrl[col][0] ^ Cntrl[col][1],
                              mux (Cntrl[col][0], B[i][0], B[i][1]),
                              ZERO);
```

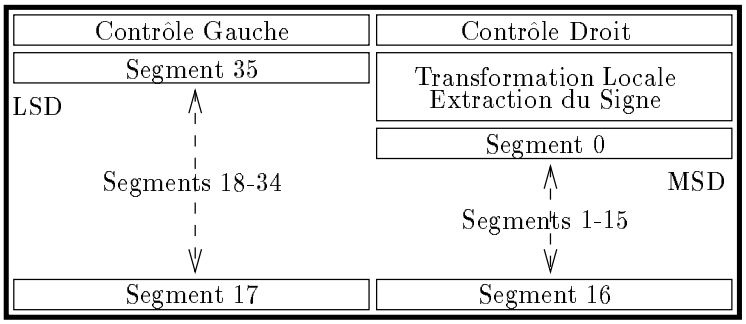


FIG. 2.15 - *Organisation des Segments dans un Circuit*

```
  AddB [i][1]      = reg (mux (Cntrl[col][0] ^ Cntrl[col][1],
                              mux (Cntrl[col][0], B[i][1], B[i][0]),
                              ZERO));
  AddB [i][0]     <= LOC (col, line);
  AddB [i][1]     <= LOC (col, line);
  AddB [i][0]     <= CLB_INPUT (B, Cntrl[col][0]);
  AddB [i][1]     <= CLB_INPUT (C, Cntrl[col][1]);
}
```

Les cellules de la première ligne de chaque circuit génèrent les signaux que nous allons détailler et qui seront émis en direction de tous les segments. Les segments d'un même circuit sont alignés sur deux colonnes afin de partager les signaux le long des lignes de diffusion verticales comme présenté dans la Figure 2.15.

Afin d'incorporer le registre d'état St_k dans le segment sans augmenter la taille du segment par rapport à ce que nous avons montré, nous avons groupé la multiplication des chiffres $a'_j \times A_k^{(j)}$ et la première cellule PPM (voir la zone encadrée de la Figure 2.14). Les deux fonctions de cette cellule groupée ont exactement cinq entrées. Ainsi, chacune peut être implantée sur une seule cellule logique en utilisant une seule des deux fonctions disponibles. L'économie vient du fait que nous avons supprimé la génération des signaux intermédiaires. La place libérée dans la première cellule est utilisée pour stocker et propager le compteur d'état. La partie libérée de la seconde cellule est utilisée pour améliorer la propagation du signal de remise à zéro. Le signal de remise à zéro intervient dans de nombreuses cellules, il faut donc le faire émettre par un peigne de diffusion pour qu'il soit accessible dans tout le circuit. Le seul signal qui bénéficie d'une telle répartition est le signal d'horloge. Pour transmettre le signal de mise à zéro à tous les registres qui l'emploient dans le circuit sans ralentir la vitesse de travail du circuit, nous avons introduit un cycle de retard en faisant passer le signal par un registre. Le signal est généré avec un cycle d'avance dans le commutateur pour conserver un comportement normal. Il est stocké dans chaque ligne du circuit, puis émis durant le cycle d'après en direction des registres sur une ligne de diffusion horizontale pour chaque ligne de cellules. La fonction arithmétique, qui est calculée par la seconde cellule que nous modifions, ne fait pas intervenir la magnitude de a'_i si l'on code le nombre sous la forme d'un chiffre signé $d = (-1)^s \times m$. Une seule ligne de diffusion verticale est donc utilisée. La seconde permet la transmission rapide du signal de pré-remise à zéro à tous les segments. La Figure 2.16 reprend l'utilisation définitive de chaque cellule du segment avec l'affectation des lignes de diffusion verticales et horizontales.

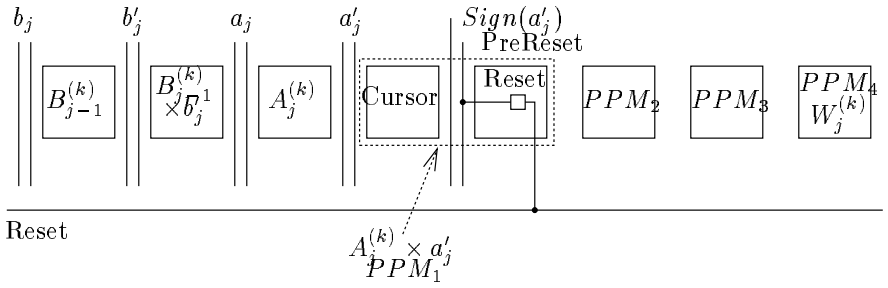


FIG. 2.16 - *Utilisation des Lignes de Diffusion par un Segment*

Nous pouvons à l'aide de cette description construire le code qui génère une cellule Nacel sur un FPGA. Les bus sont définis dans un premier temps. Puis on définit un constructeur qui va mettre en place les segments et les cellules de contrôle en fonction de la taille et de la position de la cellule que nous implantons. La fonction `Column` génère tous les segments d'une colonne ; elle est appelée par la fonction `Heart` qui provoque si besoin est deux appels pour les deux colonnes que l'on peut inclure dans le circuit. Cette fonction, ainsi que la fonction `Control` sont appelées par la fonction `Matrix` qui configure ainsi l'ensemble des cellules nécessaires.

```
class Nacel : public ChipNode {

protected:
  Digit                  Ain;          // New digit of A
  Digit                  Bin;          // New digit of B
  WireVector<Digit, 16>  Cntrl;        // Broadcast signals
  Bool                   Reset;        // Reset Signal
  WireVector<Bool , 20>  R_Reset;      // Retimed Reset
  Bool                   RR_Reset;     // Twice Retimed Reset
  DynVector <Bool >      State;        // State shift register
  DynVector <Digit>      A;            // Current value of A
  DynVector <Digit>      Aplus;        // A + Ain
  DynVector <Digit>      AddA1;        // Aplus * A'in
  DynVector <Digit>      B;            // Current value of B
  DynVector <Digit>      Bplus;        // B + Bin
  DynVector <Digit>      AddB;         // B * B'in (and not Bplus)
  DynVector <Digit>      AddAB;        // Temp for SumAB
  DynVector <Digit>      SumAB;        // (Aplus * A'in) + (B * B'in)
  DynVector <Digit>      Temp;         // Temp for Acc
  DynVector <Digit>      Sum;          // SumAB + Acc
  DynVector <Digit>      Acc;          // Accumulated shifted sum

  Nacel (char *name, int size, int first, int line1, int line2);

  void Matrix      ();

  int Size, First, Line1, Line2;
```

```
  void Input       (const int port, Bool &net);
  void Output      (const int port, Bool &net);

  void Dump                (Bool &net);

  void Cell_Aplus          (int i, int col, int line          );
  void Cell_AddA1          (int i, int col, int line          );
  void Cell_AddA2          (int i, int col, int line, int left);
  void Cell_B              (int i, int col, int line          );
  void Cell_AddB           (int i, int col, int line          );
  void Cell_SumAB          (int i, int col, int line          );
  void Cell_Temp           (int i, int col, int line          );
  void Cell_Acc            (int i, int col, int line          );

private:
  void AccountIn           (EquationHandler Input, int col, int par);
  void AccountIn           (Bool           &Input, int col, int par);
  void RepeatIn            (Digit          &Input, int col);

  void Control ();

  void Column              (int i, int col, int line, int left);
  void Heart               ();
};
```

Multiplieur Dans le multiplieur, l'information se déplace du côté gauche vers le côté droit de chaque segment (voir Figure 2.14) et du segment le moins significatif vers le segment le plus significatif (voir Figure 2.15). Aucun signal n'est émis dans la direction opposée. En introduisant une barrière temporelle de registres entre deux parties du circuit, nous créons un changement de temps entre ces deux parties. Comme les signaux sont tous émis dans la même direction, il suffit qu'aucun signal n'évolue d'une partie en retard du circuit vers une partie en avance. On peut aisément introduire deux barrières dans chaque segment et une barrière supplémentaire dans la partie du circuit qui calcule les chiffres p_j du produit.

Un circuit Xilinx entièrement occupé par des segments du multiplieur est capable de produire 74 chiffres corrects du résultat en base 2. Nous utilisons le mécanisme d'héritage en C++ pour ne définir que ce qui est ajouté par rapport à la cellule Nacel. Le multiplieur est défini comme ceci :

```
class OnLineMultiplier : public Nacel, protected Grid {

public:
  Digit                Out;           // The output digit
  Bool                 R3_Reset;      // Twice Retimed Reset
  Digit                SumOp;
  Digit                AccOp;
  WireVector<Digit,  3> T_Add;        // Temporary for OnLineAdder
```

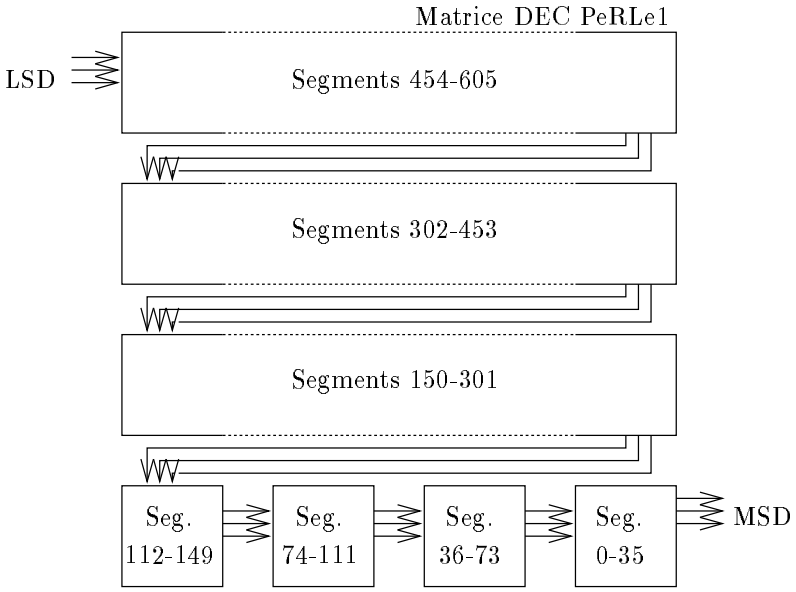


FIG. 2.17 - *Circulation des Données du Multiplieur sur la Carte PeRLe 1*

```
  OnLineMultiplier (char *name, int pos);

  void logic       ();
  void placement   () {};

protected:
  void Comm        ();
  void Surround    ();

};
```

Il est possible d'étendre le multiplieur à n'importe quelle partie de la matrice de calcul, jusqu'à 16 circuits Xilinx, afin de calculer des produits avec une grande précision. Passer d'un circuit à un autre fait intervenir comme nous l'avons vu des délais substantiels. Pour les masquer, nous avons ajouté deux barrières temporelles, cela n'est possible que parce que l'information se déplace du segment le plus significatif vers le segment le moins significatif. Le curseur d'état se déplace lui dans la direction opposée et pour qu'il arrive à temps, il est généré avec quatre cycles d'avance. Le signal peut ainsi traverser ces deux barrières temporelles et récupérer les deux cycles d'écart qui existent entre les deux circuits. Afin d'utiliser les connexions de la matrice de calcul au mieux, nous avons routé les signaux dans chaque ligne de la gauche vers la droite. Le dernier circuit à droite transmet ses données au premier circuit à gauche de la ligne suivante (voir Figure 2.17).

Nous présentons Figure 2.18 l'état du circuit 8 au cycle 322. Chaque carré représente l'état d'un registre d'une cellule logique. Les carrés clairs représente l'état actif (1), et les carrés sombres l'état passif (0). Comme les registres du multiplieur ont 605 chiffres de long, au cycle 322 seule une partie du multiplieur est active. Le circuit 8 est placé sur le bord droit de la matrice. Il est donc chargé de récupérer les données en provenance du circuit 9 et de les envoyer au circuit 12. C'est pour cela

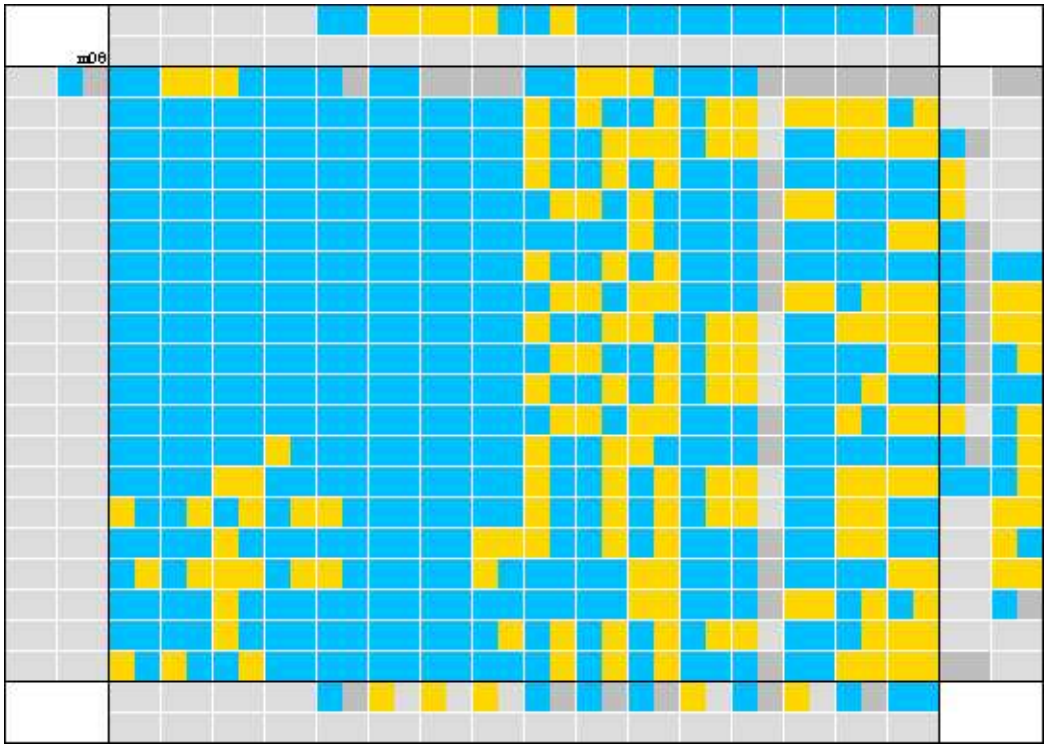


FIG. 2.18 - *Utilisation du Circuit 8 en début de Calcul*

que les connexions sur la face Sud et le bas des connexions Ouest sont utilisées.

Les seize circuits de la matrice ainsi que les sept circuits de communication avec l'hôte au cycle 650 sont représentés Figure 2.19. On constate bien que l'activité des circuits périphériques est extrêmement réduite : ils sont uniquement chargé de l'acheminement des données sans effectuer aucun traitement. Pour valider notre implantation, nous avons fait fonctionner le multiplieur en-ligne pendant plusieurs jours en vérifiant sur la machine hôte la validité des calculs.

Le routage compliqué des informations au niveau de la carte est nécessaire parce que les circuits Xilinx ne sont pas entièrement symétriques. Ils sont fortement orientés vers la droite et un peu vers le bas. Les plots d'entrée des CLB se situent sur le coté gauche, et le plots de sorties sur le coté droit, il est donc naturel de faire évoluer le signal de la gauche vers la droite. De plus, il est plus facile d'établir une connexion entre la sortie d'un CLB et l'entrée du CLB qui est directement en dessous à droite que d'atteindre celui qui est au dessus à droite. La Figure 2.20 présente le câblage du multiplieur en-ligne pour des nombres de 74 chiffres. Au niveau d'intégration que nous avions atteint, il était impossible de faire circuler les données de droite à gauche car les configurations n'avaient pas pu être implantées.

Le circuit de multiplication, que nous avons étudié ici en détail est capable d'atteindre une fréquence d'horloge de 30 MHz avec trois barrières temporelles. Sans modification dans la méthode de conception, nous pensons pouvoir ajouter deux barrières de temps supplémentaires. Mais l'on ne peut pas garantir la fréquence que l'on atteindrait alors : avec une fréquence d'horloge de l'ordre de 40 MHz, cela nous placerait trop près des performances normales de la machine. Sachant que le circuit fait intervenir de nombreuses parties très différentes de la carte PeRLe, on peut penser que cette fréquence ne serait pas atteinte. Avec une telle vitesse d'horloge, le placement et le routage devraient être entièrement effectués à la main. Les outils de développement Xilinx disponibles à

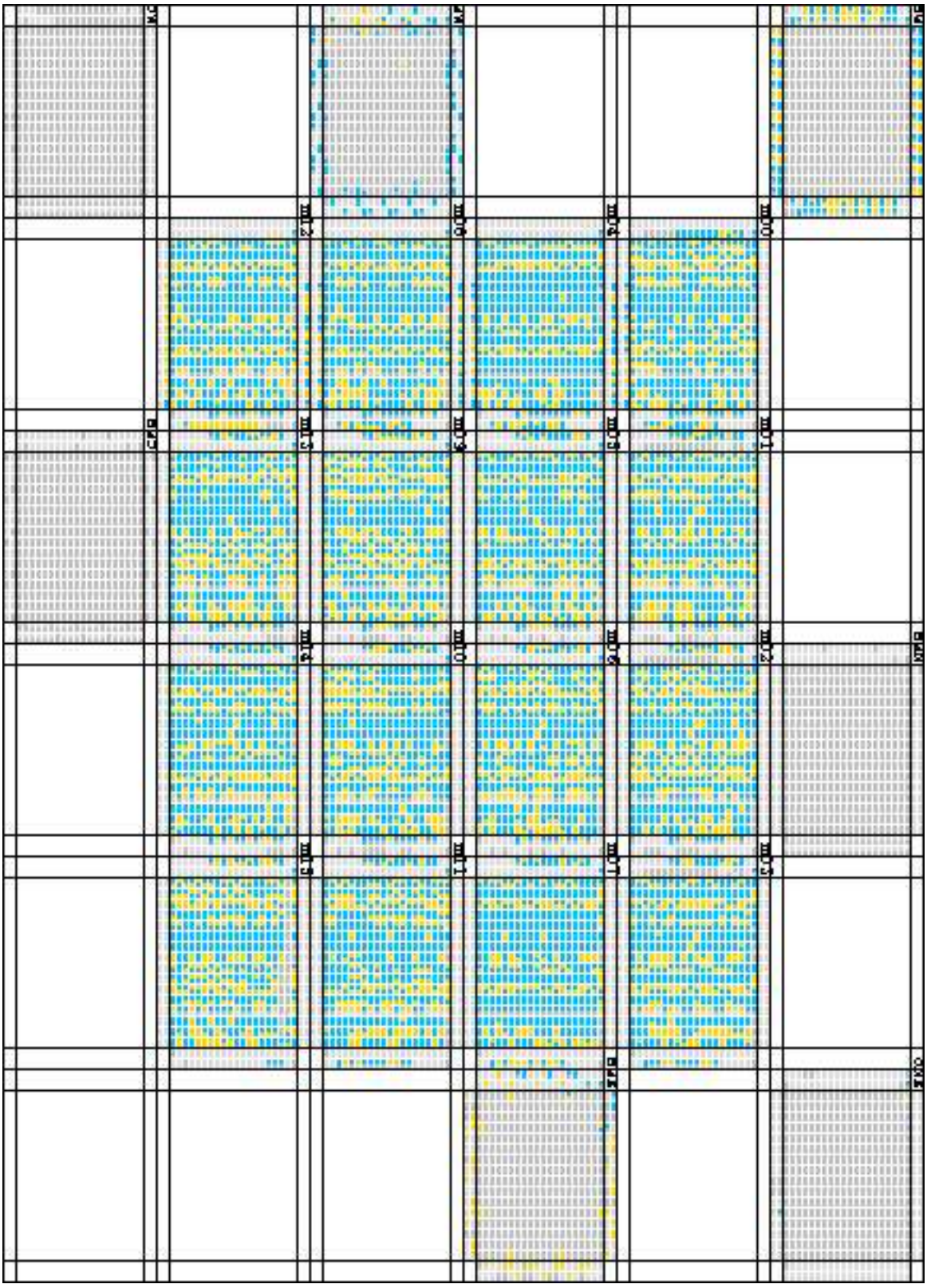


FIG. 2.19 - *Multiplication de Nombres de 1210 Chiffres sur Carte Perle*

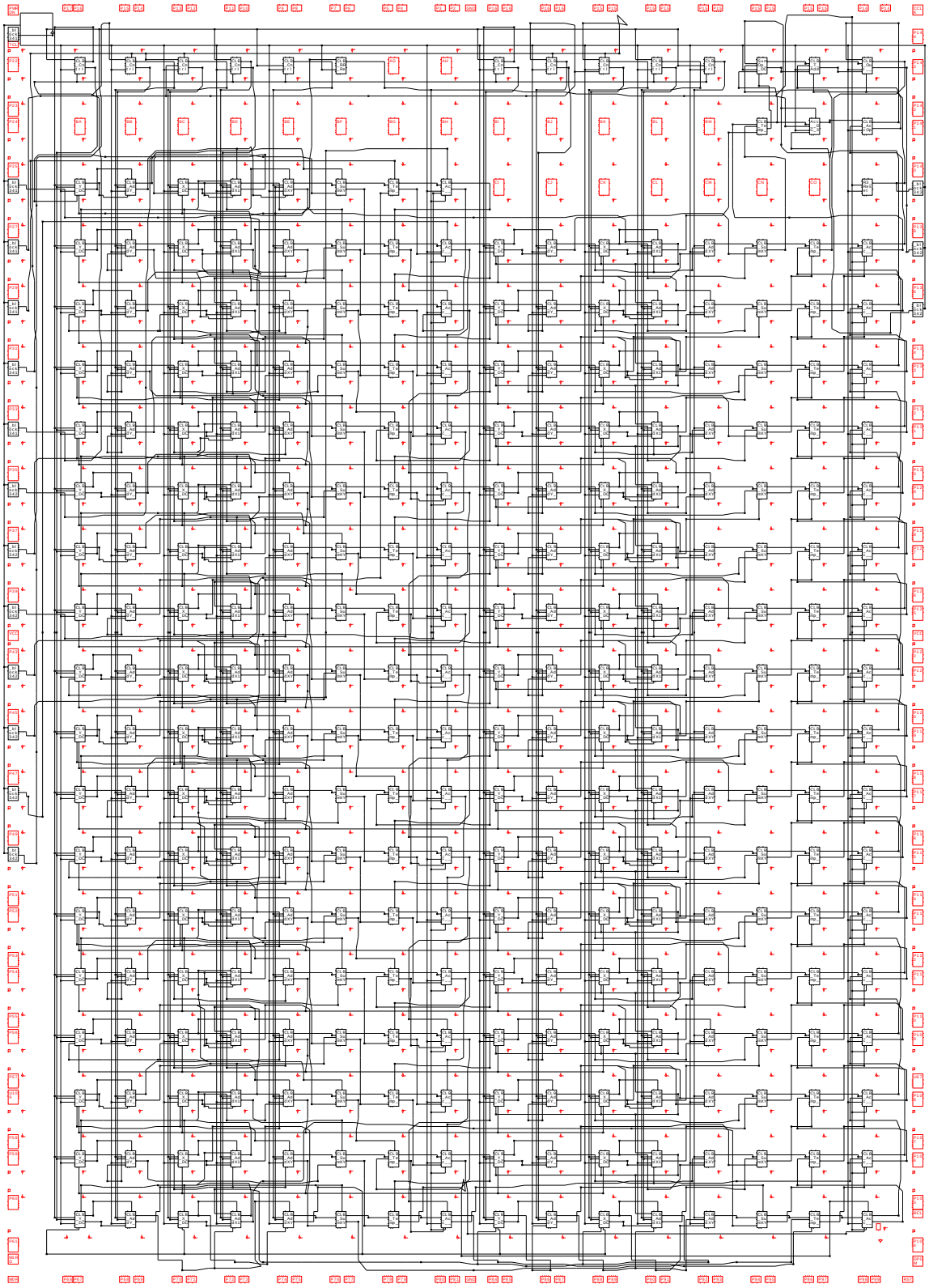


FIG. 2.20 - *Plan de Câblage du Circuit 15*

ce jour sur la carte PeRLe n'ont pas les possibilités suffisantes pour réaliser avec souplesse un tel circuit à partir de la bibliothèque PeRLe1DC.

Division et Racine Carrée Dans l'opération de division et d'extraction de racine carrée, le chiffre q_j est produit par la tête de l'opérateur et il est utilisé au même cycle dans le calcul de $q_j \times D_j$. De ce fait, il est impossible d'introduire un décalage dans le temps entre les différentes parties du circuit. Les opérateurs de base qui emploient la cellule Nacel sans aucun pipeline peuvent être utilisés avec des fréquences d'horloge relativement faibles (10 MHz).

Nous avons modifié l'algorithme pour *prédire* le chiffre suivant du quotient avec un cycle d'avance grâce à la loi de récurrence suivante. Le chiffre prédit peut alors remonter une barrière temporelle, ce qui nous permet d'atteindre une vitesse d'horloge de 20 MHz , le signal le plus long met en jeu dix cellules.

$$\begin{aligned} X_j &= X_{j-1} + x_j 2^{-j} \\ D_j &= D_{j-1} + d_j 2^{-j} \\ q_{j+1} &= sign_2(2 \times (2W_{j-1} - q_j D_3)) \\ W_j &= 2W_{j-1} - q_j D_j + d_j Q_{j-1} + \frac{x_j}{64} \\ Q_j &= Q_{j-1} + q_j 2^{-j} \end{aligned}$$

Théoriquement, on peut utiliser cette technique pour prédire le chiffre du quotient avec autant d'avance qu'on le souhaite. Mais pour chaque cycle d'avance que l'on implante, on augmente le délai pratique de l'opérateur de un. Plus important, la prédiction de q_j nécessite l'adjonction d'un nombre important de cellules logiques supplémentaires alors que l'extraction du signe était très simple pour les opérateurs normaux. Pour obtenir des fréquences d'horloge équivalentes à celles du multiplieur, on peut penser calculer q_j avec au moins trois cycles d'avance, mais cela réduirait la taille du noyau Nacel de quelques segments.

De la même façon, nous pouvons envisager le calcul d'un quotient ou d'une racine carrée sur plusieurs circuits Xilinx. À cause des barrières temporelles entre chaque circuit, pour calculer sur un peu moins de 75 chiffres, il faudrait prévoir q_j avec au moins sept cycles d'avance. Chaque circuit supplémentaire augmentera de 4 cycles l'avance nécessaire. La prédiction de signe devenant de plus en plus complexe, on ne peut raisonnablement pas espérer l'implanter pour des valeurs supérieures à cinq. Nous nous sommes donc restreint dans la pratique à des opérateurs sur 72 chiffres avec une vitesse d'horloge de 20 MHz.

Nous avons regroupé dans le tableau Figure 2.21 les caractéristiques des opérateurs que nous avons envisagés. Pour chaque opérateur, nous présentons trois implantations. La première implantation, est celle obtenue directement avec la cellule Nacel sans étude approfondie. La seconde représente ce que nous sommes capables de faire de mieux. Dans la troisième, nous présentons des résultats prospectifs en fonction des algorithmes que nous avons utilisés : il s'agit de la limite qu'il est illusoire d'espérer dépasser avec la technique de la cellule Nacel sur des circuits comparables aux Xilinx 3090. La carte PerLe-1 utilise une technologie de FPGA deja relativement ancienne, et dont les performances sont aujourd'hui depassees.

2.4 Petite Unité de Calcul En-ligne

Les prototypes et les simulateurs qui ont été réalisés dans le passé n'ont pas permis de montrer un intérêt pour les machines à flot de données universelles, et ce à cause des problèmes liés au traitement des matrices de grande taille [AC 86]. Même après la construction de quelques prototypes, le modèle d'exécution de Von Neumann n'est en rien remis en cause. Dans le domaine des architectures nouvelles, de nombreuses idées peuvent ne pas aboutir parce qu'elles sont apparues trop tôt. Dans

Opérateur		Taille des nombres	Vitesse de l'horloge
Implantation naïve	Multiplication	72	10 MHz
	Division	72	10 MHz
	Racine Carrée	72	10 MHz
Implantation évoluée	Multiplication	1210	30 MHz
	Division	72	20 MHz
	Racine Carrée	72	20 MHz
Limite technique	Multiplication	1210	$\approx$ 40 MHz
	Division	$\approx$ 140	$\approx$ 20 MHz
	Racine Carrée	$\approx$ 140	$\approx$ 20 MHz

FIG. 2.21 - *Implantation des Opérateurs En-ligne à l'aide de Nacel*

ce cas, les idées sont reprises plus tard sous une forme épurée quand la technologie est capable de les metre en œuvre. En regard des performances atteintes à ce jour par les microprocesseurs commerciaux [SM 95] on peut penser qu'une machine universelle à flot de donnée ne sera jamais capable de concurrencer les machines usuelles.

Les recherches menées sur les machines à flot de données sont à la base de nombreux résultats théoriques sur le parallélisme qui ont débouché sur des réalisations matérielles ou logicielles. Nous disposons depuis peu au sein de l'équipe SAAO d'un processeur Pentium Pro d'Intel en vue de réaliser des tests fonctionnels. Ce processeur utilise une technique de table de reservation pour garantir le fonctionnement à plein régime du pipeline d'exécution [Gwe 95]. Le pipeline d'instruction induit une forme de parallélisme qui peut provoquer sur les registres des conflits suite à des aléas d'écriture après lecture ou d'écriture après écriture. Les compilateurs ne prennent pas en compte ces problèmes qui dépendent directement de l'architecture du processeur. Grâce à la technique des tables de reservation, on peut réduire énormément le nombre de cas où il faut insérer une instruction vide dans le pipeline afin de supprimer un aléa. Ces ressources sont néanmoins inaccessibles à l'utilisateur. La notion de processus leger qui est maintenant répandue sur tous les systèmes unix a aussi été utilisée et mise en avant par le paradigme d'exécution à flot de donnée, car elle permet en regroupant des morceuax de programme à la compilation de tirer parti des microproesseurs très puissant disponibles maintenant pour concevoir des machines hybrides. Nous avons vu aussi apparaître les langages *Id* et *Sisal* beaucoup plus adaptés aux machines parallèles [MGr 83, AGP 78]. Le compilateur *OSC* [FCO 90] permet d'exécuter un algorithme sur une machine parallèle (Cray, Sequent, Convex *etc...*) sans que le programmeur ait besoin de connaître les spécificités de l'architecture de la machine [DE 93]. L'unité Puce dont je vais décrire maintenant les fonctionnalités est basée sur le prototype de la *Manchester DataFlow Machine* (MDM). Cette machine offre les fonctions usuelles de contrôle (exécution conditionnelle, boucle *etc...*) et toute la généralité nécessaire à l'évaluation d'expressions numériques compliquées.

2.4.1 Manchester DataFlow Machine

La Manchester DataFlow Machine (MDM) est un prototype de machine à flot de données qui a été achevé en 1981. Un programme est représenté par un graphe dirigé sans cycle (*Directed Acyclic Graph*). Les données sont introduites au sommet de l'arbre. Chaque fois qu'un opérateur dispose

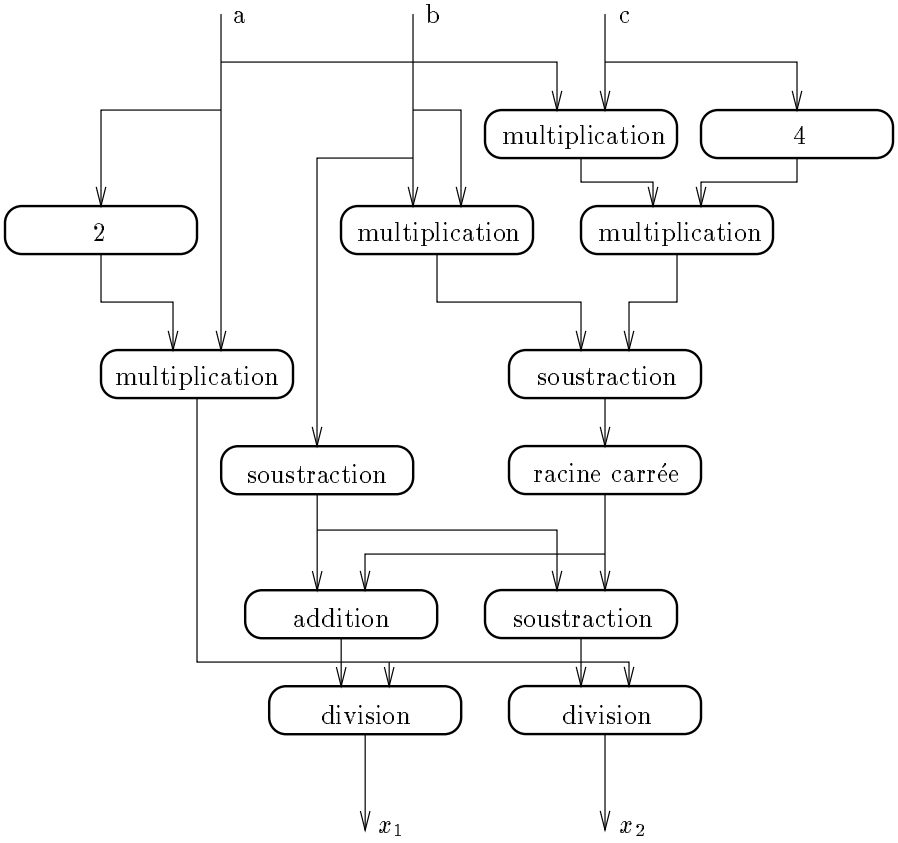


FIG. 2.22 - *Graphe de Calcul des Racines d'un Polynôme du Second Degré*

de tous ses opérandes, il est mis à feu (*fired*), puis il produit des résultats qui sont propagés aux opérateurs suivants. Le résultat du programme se trouve dès qu'il est disponible à la racine du graphe (voir Figure 2.22).

Dans une machine statique, chaque nœud du graphe est associé à une instruction en mémoire. L'instruction stocke avec son code d'opération les opérandes qui sont en attente et l'adresse des nœuds auxquels elle doit envoyer le résultat. On ne peut pas mettre en commun deux parties de code identiques, le code n'est pas réentrant. Il faut alors dérouler toutes les boucles du programme à la compilation.

Dans une machine à jeton étiqueté comme la *Tagged Token DataFlow Machine* ou la MDM, les opérandes ne circulent plus seuls, ils sont encapsulés dans un jeton qui contient une variable définissant l'environnement. Deux itérations différentes d'une boucle peuvent partager le même code, il suffit pour cela qu'elles aient un environnement différent. Le champ `index` de l'environnement de la MDM est chargé de retenir l'indice de l'itération à laquelle appartient le jeton. La destination d'un jeton n'est plus une instruction, mais un couple constitué de l'instruction et de l'environnement.

Les opérateurs de calcul n'agissent en général pas sur l'environnement, mais un appel de fonction ou un opérateur de construction de boucle génèrent un nouvel environnement. Chaque nœud stocke l'adresse des opérateurs à qui il doit envoyer ses résultats sauf pour un retour de fonction dont la destination est déduite de l'environnement que le programme quitte. On obtient sur une machine

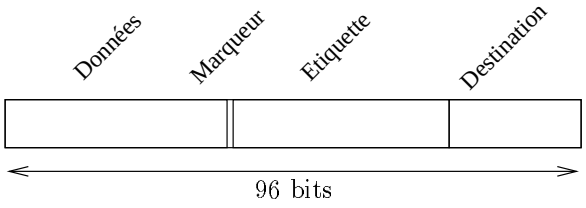


FIG. 2.23 - *Jeton de la Manchester DataFlow Machine*

à jeton étiqueté une souplesse d'utilisation comparable à celle d'une machine de Von Neumann sauf pour le traitement des structures de données : en toute généralité pour mettre à jour une cellule d'une matrice, il faut faire passer l'ensemble de la matrice à travers les opérateurs de calcul. Néanmoins, pour certains problèmes comme le pivot de Gauss, une bonne écriture des algorithmes spécialement adaptées aux machines à flot de donnée permet de contourner le problème.

Il n'est plus possible de stocker les jetons en attente au fur et à mesure qu'ils arrivent avec le code d'opération de l'instruction, car il ne suffit pas d'avoir un jeton pour chaque opérande pour pouvoir mettre à feu l'opérateur. On se restreint volontairement à des instructions unaires ou binaires. Dès qu'une donnée est disponible pour une instruction unaire, celle-ci peut être mise à feu, et on n'a jamais besoin de stocker ce jeton. Pour les opérateurs binaires, il faut apparier les jetons qui ont pour destination la même instruction dans le programme et le même environnement, en conservant en attente les jetons qui ne sont pas encore appariés. On implante ce processus à l'aide d'une mémoire associative ou à partir d'un dispositif qui se comporte comme une mémoire associative. La destination et l'environnement constituent la clef de cette mémoire. Dans le cas de la MDM, la clef a 58 bits de long, et le dispositif est capable de stocker 1 Mmots. Pour cela, les concepteurs ont eu recours à une table de hachage avec un niveau de gestion des collision, et une mémoire annexe de débordement. Un jeton MDM est composé des quatre champs présentés Figure 2.23.

Les jetons des données du programme sont placés initialement dans la mémoire associative. Dès que deux jetons sont disponibles pour le même opérateur et avec le même environnement, ils sont envoyés dans la file d'attente vers les unités fonctionnelles. Les jetons qui sont destinés aux opérateurs unaires sont mis dans la file d'attente avec un jeton fantôme dès qu'ils sont générés. À chaque cycle l'unité fonctionnelle retire un couple de jetons de la file d'attente et le traite. Elle génère un ou deux jetons qui sont envoyés à la mémoire associative au travers d'une file d'attente. On peut implanter jusqu'à 20 unités fonctionnelles sur une même carte. Tant qu'il y a assez de jetons dans le système pour maintenir toujours un couple dans la file d'attente, la latence du réseau et de la mémoire associative ne ralentit pas le traitement. Nous avons présenté un schéma de fonctionnement d'un unité fonctionnelle de la MDM Figure 2.24.

Dans [GKW 85], les auteurs présentent le résultat de tests sur 14 applications avec 29 entrées différentes. Les applications sont codées pour la plupart en Sisal qui est un langage de haut niveau comparable aux langages utilisés actuellement. La machine présente dans sa configuration maximale avec 12 processeurs une efficacité entre 8 et 9 avec une puissance de calcul comprise entre 1 et 1.5 MIPS.

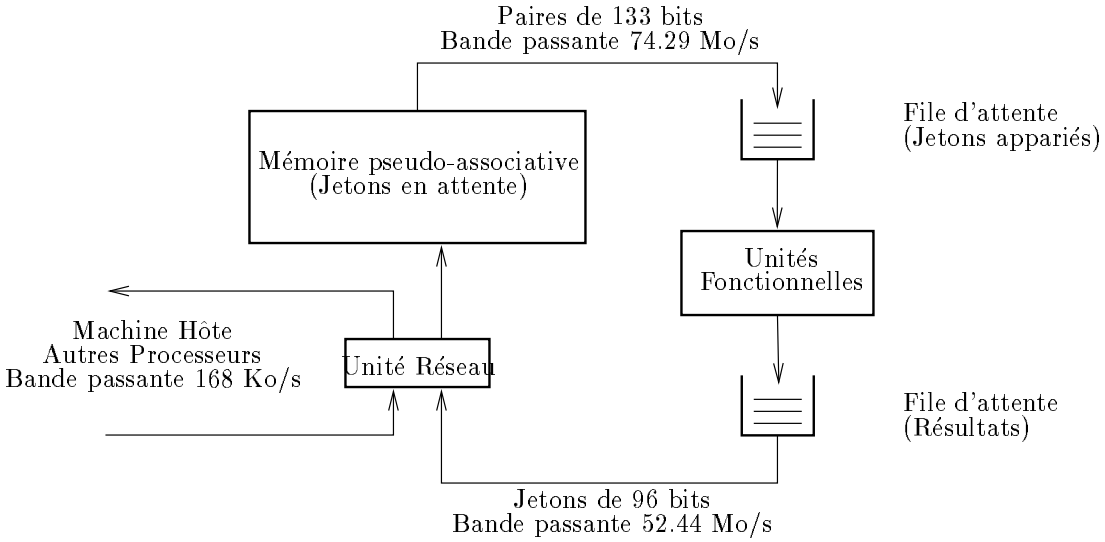


FIG. 2.24 - *Schéma de Fonctionnement de la Manchester DataFlow Machine*

2.4.2 Machine Virtuelle Puce

Nous avons conçu la machine virtuelle Puce à partir du paradigme de fonctionnement de la MDM. Cette unité peut prendre en charge l'évaluation d'expressions simples : multiplication de matrices, relaxation, *etc*... Utilisée en tant que processeur numérique étendu elle retourne le résultat sous la forme IEEE normalisée [IEEE 85] en fournissant une borne explicite de l'erreur. Une analyse du contenu de la mémoire associative indique à tout moment les données dont la précision doit être affinée pour continuer le calcul. La mise en place d'une nouvelle expression ne requiert que l'effacement de la mémoire associative, les chargements des jetons des données et de l'arbre d'évaluation.

Les champs du jeton sont adaptés à l'unité virtuelle : ainsi un petit champ de donnée permet plus de souplesse à l'utilisation et conserve la dynamique de précision de l'arithmétique en ligne. De plus, le calcul en-ligne favorise les petites bases à cause de l'initialisation des opérateurs : dans une petite base, on perd moins d'information si l'opérateur doit conserver un chiffre pour stocker son état intermédiaire. Par contre, il faudra réaliser plus d'opérations élémentaires pour obtenir le même nombre de chiffres du résultat. À cause du temps nécessaire à la génération et à l'appariement des jetons, il est essentiel de ne pas prendre une base trop petite comme 2. L'écriture des nombres en base 10 ou 16 représente un bon compromis qui pourra être exploré davantage par la suite. Les choix ne seront probablement pas les même dans une bibliothèque optimisée et prête à l'emploi ou dans une implantation matérielle. Les champs qui composent un jeton sont définis Figure 2.25.

Soit un jeton `T`. On accède à ses différents champs grâce aux mots-clef `Val` pour la donnée, `Exp` pour le champ exposant, `Itér` pour le champ itération, `Env` pour le reste de l'environnement. Pour définir les instructions nous allons utiliser trois primitives. `Recopie` crée une copie du jeton dans un autre jeton, `Modifie` change la valeur des champs indiqués sans affecter les autres champs, et `Envoie` met à jour le champ destination du jeton.

Des macro-opérateurs logiques (tests, branchements conditionnels *etc*...) permettent de traiter les nombres en-ligne dans toute leur généralité. Les opérateurs de contrôle et de manipulation de l'environnement que nous définissons pour le calcul d'un produit ou d'une somme sont décrits ici.

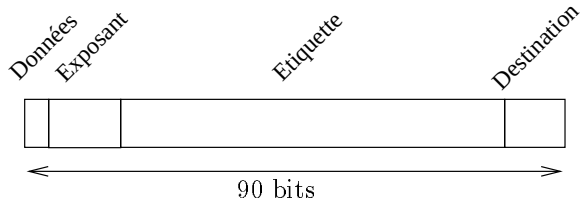


FIG. 2.25 - *Jeton du Prototype Logiciel d'Unité Puce*

Nous avons également implanté un opérateur de pseudo-normalisation et un opérateur d'itération, mais dans un souci de concision, nous nous limiterons aux instructions qui sont utilisées pour la transformée de Fourier rapide présentée en exemple dans la fin de ce chapitre.

Duplique `DUP` reproduit intégralement le jeton qu'elle reçoit en entrée sur les deux ports de sortie. Aucune modification n'est effectuée. Il s'agit de la même opération que celle implantée sur la MDM. Afin de limiter la taille des instructions en mémoire, un opérateur ne peut avoir que deux sorties. Si une donnée doit être envoyée à plus d'opérateurs, on la duplique à l'aide d'un arbre de `DUP`.

`DUP` (T1) $\longrightarrow$ port1, port2

```
Recopie T1 dans TA
Recopie T1 dans TB

Envoie TA à port1
Envoie TB à port2
```

Empile `PUSH` génère un nouvel environnement où l'exposant est l'indice d'itération. L'instruction positionne l'exposant à 0 dans le jeton résultant. Si deux opérateurs sont connectés (deux adresses de sortie), le jeton est dupliqué pour être envoyé à chaque opérateur. L'algorithme de gestion des environnements conditionne pour une bonne part l'efficacité de la machine :

- Le mécanisme doit être distribué afin que l'on ne soit pas obligé de recourir à un unique serveur d'environnement. Il faut aussi qu'il soit rapide et bijectif.
- Si deux jetons qui possèdent le même environnement sont consommés par un `PUSH`, les deux jetons créés doivent également avoir le même environnement. Dans le cas contraire, on ne peut pas garantir que les différents chiffres d'un nombre empruntent le même chemin.
- Pour pouvoir évaluer des expressions complexes, il faut être capable d'imbriquer quelques boucles (deux niveaux) ou de faire des appels de fonction. Comme la plupart des opérateurs arithmétiques en-ligne utilisent eux-mêmes un niveau de l'environnement pour leurs calculs internes, cela nécessite la possibilité d'imbriquer quatre niveaux.
- La taille des boucles qui seront exécutées est limitée par la valeur maximale du champ `Itér`. Il faut aussi que le champ `Itér` puisse être facilement extrait et modifié car de nombreuses instructions y font appel.

– L'environnement est pris en compte par l'unité d'appariement des jetons avec l'instruction cible. Si on implante cette unité à l'aide d'une mémoire associative, il est nécessaire de restreindre la taille de la clef. Même si on implante l'unité d'appariement à l'aide d'une table de hachage, il est utile de limiter la taille du champ environnement pour limiter la taille des jetons et des chemins de données.

Le mécanisme que nous avons mis en œuvre dans le prototype logiciel de Puce est très simple. Il définit l'environnement sur 48 bits, les douze bits de poids faible sont réservés au champ `Itér`. La génération d'un nouvel environnement est réalisée par un décalage de 12 bits de la variable environnement vers la gauche. Cela limite le prototype logiciel à des boucles de taille 4096 et à 4 appels de fonction ou boucles imbriquées.

PUSH (T1) —→ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Exp  = 0
                Itér = Exp (T1)
                Env  = Nouvel Env (Env (T1), Itér (T1))

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Dépile `POP` est utilisée pour retourner dans l'environnement précédent sur la pile. De plus, elle initialise l'exposant avec l'indice d'itération de l'environnement sortant.

POP (T1) —→ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Exp  = Itér (T1)
                Itér = Extrait Itér (Env (T1))
                Env  = Dépile  Env  (Env (T1))

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Déroule `UNROLL` est une instruction de contrôle de boucle qui décrémente l'indice d'itération du jeton. Les deux instructions `UNROLL` et `ROLL` jouent le même rôle que les instructions `ADL` et `SIL` de la MDM, respectivement *add to iteration level* et *set iteration level*. Nous avons voulu montrer dans cette étude l'intérêt d'une machine en-ligne à flot de données. Aucun travail n'a encore été réalisé pour réduire et adapter au mieux le jeu d'instructions. Il est probable que dans l'avenir, ces deux instructions laisseront la place à une seule instruction commune.

UNROLL (T1) —→ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Itér = Itér (T1) - 1

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Boucle `ROLL` incrémente l'indice d'itération du jeton.

ROLL (T1) $\longrightarrow$ *port1, port2*

```
Recopie T1 dans TA
Modifie TA avec Itér = Itér (T1) + 1

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Échange `SWAP` échange la valeur de l'exposant et de l'indice d'itération. Cette instruction permet de faire agir un jeton sur tous les chiffres d'un nombre comme c'est le cas dans le produit d'un nombre par un chiffre. En conséquence, le champ `Itér` et le champ `Exp` d'un jeton ont la même taille. Il est normal que ces deux champs jouent un rôle symétrique parce que `Exp` sert à coder l'indice d'un chiffre dans le nombre flottant en-ligne, et nous utilisons des boucles pour calculer sur tous les chiffres d'un mot, il faut donc que les boucles puissent au moins être aussi longues que le nombre de chiffres dans un nombre flottant en-ligne.

SWAP (T1) $\longrightarrow$ *port1, port2*

```
Recopie T1 dans TA
Modifie TA avec Exp  = Itér (T1)
                Itér = Exp  (T1)

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Saut `JUMP` effectue une multiplication d'une puissance de la base de numération en ajoutant à l'indice du jeton reçu sur le port droit l'exposant du jeton reçu sur le port gauche et en conservant les champs de données inchangés. Il s'agit ici d'un saut d'une itération à une autre. Le jeton T2 est modifié afin de pouvoir agir avec le chiffre suivant du mot tout en conservant la même valeur du saut.

JUMP (T1, T2) $\longrightarrow$ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Itér = Itér (T1) + Exp (T2)

Recopie T2 dans TB
Modifie TB avec Itér = Itér (T2) + 1

Envoie TA à port1
Envoie TB à port2
```

Les opérations arithmétiques de Puce modifient la valeur du champ exposant ou du champ donnée. Nous allons voir tout d'abord les opérateurs qui agissent sur les exposants.

Comparaison d'Exposants `EXP_COMP` calcule le maximum et la différence des deux exposants. Cette instruction est utilisée pour l'addition de deux nombres flottants en ligne.

EXP_COMP (T1, T2) $\longrightarrow$ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Val = 0
                Exp = max (Exp (T1), Exp (T2))

Recopie T2 dans TB
Modifie TB avec Val = 0
                Exp = Exp (T1) - Exp (T2)

Envoie TA à port1
Envoie TB à port2
```

Incrémente l'Exposant `EXP_INC` incrémente la valeur de l'exposant. Dans le cas d'une addition en-ligne, il y a possibilité d'un dépassement de capacité. L'opérateur d'addition en-ligne que nous allons décrire insère un 0 en tête de mot si le dépassement de capacité n'a pas eu lieu.

EXP_INC (T1) $\longrightarrow$ port1, port2

```
Recopie T1 dans TA
Modifie TA avec Val = 0
                Exp = Exp (T1) + 1

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Sélection d'Exposant `EXP_SELECT` reçoit sur son entrée la différence des exposants de deux nombres à ajouter dans un additionneur flottant. Il génère les jetons qui seront utilisés pour programmer les décaleurs et générer en tête de mot le nombre correspondant de zéros. Ainsi, on voit que si la différence `Exp(T1)` est positive, le premier décaleur est activé et insère `Exp(T1)` 0. Si la différence est négative, on active le second décaleur. Le décalage est obtenu à l'aide de l'opération `JUMP`.

EXP_SELECT (T1) $\longrightarrow$ *port1, port2*

```
Recopie T1 dans TA
Modifie TA avec Val = 0
               Exp = max ( Exp (T1), 0)

Recopie T1 dans TB
Modifie TB avec Val = 0
               Exp = max (-Exp (T1), 0)


Envoie TA à port1
Envoie TB à port2
```

Somme d'Exposants `EXP_SUM` calcule la somme des deux champs exposants définis dans ses entrées. Il est principalement utilisé par la macro-opération de multiplication. Nous avons ici ajouté automatiquement 2 à la valeur trouvée pour prendre en compte le possible dépassement flou qui peut avoir lieu lors d'une multiplication en-ligne. Cela n'a pas été fait pour l'addition parce que si on avait couplé `EXP_COMP` et `EXP_INC` sur un seul opérateur, il aurait fallu insérer une instruction `DUP` pour obtenir les trois jetons nécessaires.

EXP_SUM (T1, T2) $\longrightarrow$ *port1, port2*

```
Recopie T1 dans TA
Modifie TA avec Exp = Exp (T1) + Exp (T2) + 2

Recopie T1 dans TB
Modifie TB avec Val = 0

Envoie TA à port1
Envoie TB à port2
```

Les instructions qui manipulent le champ valeur des jetons sont les suivantes.

Addition avec Retenue `ADD_CARRY` calcule la somme de deux valeurs en générant un résultat et une retenue. Cette instruction est symétrique, les deux entrées jouent le même rôle. Les algorithmes dérivés de celui d'A. Avižienis calculent une somme en base `BASE` sans propagation de retenue à l'aide d'un système de numération qui contiennent seulement `BASE + 1` chiffres. Néanmoins, pour faire ce calcul, il faut construire une cellule capable d'examiner trois chiffres

pour donner un chiffre du résultat. C'est en contradiction avec le modèle qui n'autorise que des opérations unaire ou binaires. Nous utilisons un système de numération qui contient `BASE + 2` chiffres pour contourner ce problème. La base étant paire, les chiffres autorisés vont de `DEMI_BASE + 1` à `-DEMI_BASE`. Ainsi, pour être sûr d'absorber la retenue sortante de l'addition des chiffres de rang inférieur, il faut que la somme soit comprise entre `-DEMI_BASE + 1` et `DEMI_BASE`.

ADD_CARRY (T1, T2) ⟶ port1, port2

```
somme   = Val (T1) + Val (T2)
retenue = 0

Si somme  >  DEMI_BASE alors somme   = somme - BASE
                             retenue =  1
Si somme <= -DEMI_BASE alors somme   = somme + BASE
                             retenue = -1

Recopie T1 dans TA
Modifie TA avec Val  = somme
                Itér = Itér (T1) - 1

Recopie T1 dans TB
Modifie TB avec Val  = retenue
                Exp  = Exp (T1) + Exp (T2) + 1

Envoie TA à port1
Envoie TB à port2
```

Addition sans Retenue `ADD_FREE` calcule la somme de deux valeurs en supposant qu'aucune retenue ne devra être générée. Pour cela, on vérifie que l'une des entrée ne contient que des jetons générés par le port gauche d'un opérateur `ADD_CARRY` et la seconde entrée que des jetons générés par le port droit de cet opérateur.

ADD_FREE (T1, T2) ⟶ port1, port2

```
Recopie T2 dans TA
Modifie TA avec Val  = Val (T1) + Val (T2)

Recopie TA dans TB

Envoie TA à port1
Envoie TB à port2
```

Produit (Poids Fort) `PROD_HI` calcule le chiffre de poids fort du produit de deux jetons. Il est utilisé en conjonction avec `PROD_LO` dans le calcul de la multiplication d'un nombre par un chiffre. Avec une base supérieure à 5 et un système redondant relativement centré, on est

sûr de pouvoir ajouter une retenue au résultat du chiffre de poids fort sans engendrer un nouvelle retenue. L'indice d'itération du jeton TA est généré pour permettre l'accumulation des produits partiels : il s'agit d'un produit partiel de ce rang. Le jeton TB est modifié par rapport au jeton T2 pour pouvoir l'utiliser dans les prochains produits partiels.

PROD_HI (T1, T2) $\longrightarrow$ *port1, port2*

```
haut = (Val (T1) * Val (T2))  /  BASE
bas  = (Val (T1) * Val (T2)) mod BASE

Si bas  >  DEMI_BASE alors bas  = bas  - BASE
                           haut = haut + 1
Si bas <= -DEMI_BASE alors bas  = bas  + BASE
                           haut = haut - 1

Recopie T1 dans TA
Modifie TA avec Val  = haut
                Exp  = 0
                Itér = Itér (T1) + Exp (T1) + 1

Recopie T2 dans TB
Modifie TB avec Exp  = Itér (T2) + 1
                Itér = Exp  (T2)

Envoie TA à port1
Envoie TB à port2
```

Produit (Poids Faible) `PROD_LO` calcule le chiffre de poids faible du produit de deux jetons. Nous utilisons la même méthode que pour l'addition avec retenue. On peut ainsi envoyer directement les jetons produits par ces deux instructions à un opérateur `ADD_FREE` s'il faut consommer des jetons de retenue. Les jetons TA et TB subissent sensiblement le même traitement que pour l'opération qui calcule le chiffre de poids fort.

PROD_LO (T1, T2) $\longrightarrow$ *port1, port2*

```
bas  = (Val (T1) * Val (T2)) mod BASE

Si bas  >  DEMI_BASE alors bas  = bas  - BASE
Si bas <= -DEMI_BASE alors bas  = bas  + BASE

Recopie T2 dans TA
Modifie TA avec Val  = bas
                Exp  = 0
                Itér = Itér (T2) + Exp (T2) + 1

Recopie T2 dans TB
Modifie TB avec Exp  = Itér (T2) + 1
```

```
                Itér = Exp  (T2)

    Envoie TA à port1
    Envoie TB à port2
```

2.4.3 Vers une Implantation de Puce

Nous pouvons nous inspirer des travaux réalisés autour des machines à flot de données pour proposer trois implantations possibles pour une future unité Puce. Mes travaux, dans le cadre de l'unité Puce, représentent une approche originale vis à vis des recherches antérieures menées pour offrir une implantation efficace du calcul en-ligne. Même si le modèle d'exécution à flot de données peut être déroutant si l'on considère les performances actuelles des machines de Von Neumann, Puce se démarque des propositions antérieures parce qu'elle ne fait appel qu'à un jeu restreint d'instructions pour réaliser toutes les opérations, et qu'elle utilise abondamment la mémoire. Puce se rapproche ainsi beaucoup des architectures usuelles. On devra quand même mener une étude plus poussée sur un processeur matériel Puce avec une descripion de bas niveau, en VHDL par exemple, pour évaluer précisement les performances d'une telle réalisation. Nous allons détailler plutôt trois approches qui utilisent les moyens qui sont d'ores et déjà à notre disposition.

- Les microprocesseurs disponibles actuellement atteignent des fréquences d'horloge de 200 MHz ou 400 MHz avec plusieurs unités fonctionnelles et une hiérarchie de caches très efficace. Afin d'implanter une bibliothèque qui reproduit efficacement Puce, il nous faudra simuler le comportement d'une machine à flot de données. En regroupant les jetons consécutifs d'un même nombre, on peut reconstruire un début de localité dans le code qui effectue les calculs. De plus, l'absence de localité dans les machines à flot de données aura un effet très réduit car la taille des caches présentes sur un microprocesseur augmentant, on peut stocker l'ensemble des jetons d'un calcul simple sur quelques lignes de cache, même si à l'intérieur de ces pages les accés aux jetons sont imprévisibles.

  Dans le cadre d'une implantation logicielle, il est essentiel d'augmenter la taille du champ de donnée. Un champ de taille comparable à celle d'un mot machine offrira la puissance maximale de calcul. Par contre cela nuit légérement à la dynamique d'évaluation. Cette solution est à réserver aux utilisateurs qui demandent une très grande précision, de plusieurs milliers de bits, tout en conservant une bonne dynamique pour la précision de chaque calcul. Dans les cas plus courants, on pourra adopter un champ de donnée de l'ordre de 10 ou 16 bits.

  Afin de tirer parti des fonctionnalités des ordinateurs modernes, la bibliothèque devra maintenir en permanence le plus possible d'unités fonctionnelles en activité. Pour cela, il suffit de mener plusieurs calculs de façon concurrente. Cela signifie que dans des cycles entrelacés, on effectue les calculs sur les champs de donnée d'une paire, la génération de l'environnement d'une autre paire et l'appariement de jetons générés quelques cycles plus tôt. On garantit ainsi qu'il n'y aura plus de dépendance de donnée entre les instructions, que l'on paiera rarement le prix d'un accés à la mémoire centrale ou que les prédictions de branchement effectueront toujours un travail utile. Dans la mesure où pendant chaque cycle l'ensemble des unités fonctionnelles est utilisé, on peut évaluer le temps de traitement d'un jeton, τ entre 20 et 50 cycles de façon fort conservative. Néanmoins, afin de garantir qu'aucune dépendance de donnée ne ralentisse le fonctionnement de la bibliothèque d'évaluation, on se donne un intervalle de temps 10 ou 50 fois plus long pendant lequel il faut placer exactement les τ cycles de

traitement d'une paire de jetons. Avec les microprocesseurs actuels, nous pouvons avec ces hypothèses conservatives réaliser une machine virtuelle qui fonctionne à la vitesse de 10 MHz avec une latence qui se situera entre 10 cycles et 50 cycles.

- Nous apprendrons sûrement beaucoup en implantant un prototype matériel de tests de Puce. Cette étude donnera probablement des performances inférieures à celles que l'on pourrait attendre d'une machine virtuelle qui utilise à plein les capacités d'un microprocesseur moderne. Cela nous permettra de préciser les zones de contention d'une hypothétique machine Puce. Mes premières analyses montrent que les unités fonctionnelles nécessitent très peu de ressources circuit. Toutefois, la bande passante nécessaire entre l'unité d'appariement et les unités fonctionnelles est énorme et limite le parallélisme à quelques unités fonctionnelles. Afin de réduire le besoin en accès mémoire, on pourra implanter un premier niveau de cache pour les jetons à l'intérieur d'un circuit FPGA reconfigurable et de conception plus récente que les XC3090.
- Les tous derniers microprocesseurs intègrent de plus en plus de fonctionnalités dont, pour certains, des fonctions évoluées de traitement numérique du signal [Sla 95]. Avec l'augmentation du nombre de transistors que l'on peut intégrer dans un circuit, on peut espérer que la taille des tables de réservation qui sont maintenant disponibles sur le Pentium Pro, va continuer à augmenter avec les nouvelles générations de circuits. Ainsi, on peut espérer que dans un avenir un peu plus lointain, les tables de réservation ne soient plus seulement réservées pour réordonnancer les opérations afin de supprimer les aléas dûs au pipeline. On pourra aussi les utiliser pour programmer des processus hyper-légers capables de se mettre automatiquement en marche dès que ses données sont disponibles. Si cela devait être le cas, la machine Puce pourrait être programmée très facilement.

2.5 Opérateurs Arithmétiques sur Puce

Les opérateurs arithmétiques sont des macro-opérations constituées de plusieurs instructions machines qui opèrent sur des nombres flottants à précision variable décrits grâce à une liste de jetons. Chaque entrée de l'opérateur comporte un port pour les chiffres de la mantisse, jetons M, et un port pour l'exposant, jeton E. Le champ `Exp (E)` représente l'exposant du jeton M d'indice 0. Les jetons M successifs sont numérotés d'après leur ordre dans le mot, l'indice est stocké dans le champ exposant. Les valeurs sont codées en complément à 2. Un nombre X s'écrit comme suit. La base de numération est β. Dans notre cas, $\beta \in \{10, 16, 2^{16}, 2^{30}\}$.

$$X = \left(\sum_{M} Val(M) \times \beta^{-Exp(M)} \right) \times \beta^{Exp(E)}$$

Pour implanter les fonctions élémentaires de l'arithmétique en-ligne, il faut être capable de réaliser des opérateurs de taille variable [Mul 94]. Dans la suite de ce chapitre, je présente le découpage d'un opérateur de taille variable en micro-opérations de taille fixe. Des travaux comparables ont été réalisés pour l'évaluation en grande base d'expressions arithmétiques complexes sur des machines parallèles [Maz 93].

2.5.1 Définition des Opérateurs

Addition La macro-opération d'addition est constituée de 19 instructions, elle est organisée d'après la Figure 2.26. L'unité exposant (UE) manipule les jetons exposants des nombres (voir

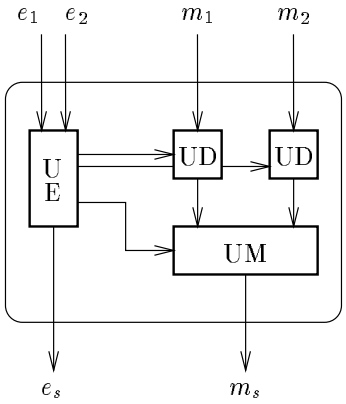


FIG. 2.26 - *Additionneur — Schéma de Fonctionnement*

Figure 2.27) : `EXP_COMP` calcule le maximum et la différence des deux exposants ; le maximum est incrémenté par l'opérateur `EXP_INC` pour obtenir l'exposant du résultat ; la valeur absolue de la différence est envoyée par l'instruction `EXP_SELECT` qui selon son signe la transmet à l'unité de décalage de la mantisse gauche ou droite.

L'unité de décalage (UD) insère des zéros en tête du nombre dont l'exposant est le plus faible avant d'effectuer l'addition des mantisses comme le fait une unité de calcul flottant. Cette unité présentée Figure 2.28 est constituée uniquement d'opérateurs de contrôle. L'unité de calcul sur les mantisses (UM), Figure 2.29, comprend quatre instructions seulement. L'opérateur `ADD_CARRY` effectue l'addition de deux nombres en générant une retenue et une somme. L'opérateur `ADD_FREE` calcule la somme sans retenue entre un chiffre et une retenue (valeur 0, 1 ou -1). L'instruction `PUSH` insère un jeton nul dans l'additionneur pour absorber la retenue de poids fort et l'instruction `POP` restaure l'environnement initial du nombre.

Multiplication L'opérateur de multiplication est construit à partir d'une petite unité de calcul sur les exposants et d'une unité de calcul sur les mantisses. L'instruction `EXP_SUM` calcule la somme des deux exposants. Comme dans le cas de l'addition, nous sommes obligés de générer un dépassement flou de capacité en augmentant de 2 la valeur de l'exposant [DM 91]. Le second jeton créé par l'instruction `EXP_INC` est utilisé par l'unité de calcul sur les mantisses pour absorber une retenue.

Les jetons de mantisse sont traités de façon à générer tous les produits chiffre à chiffre. Pour ce faire, on utilise à la fois l'indice d'itération de l'environnement et le champ exposant des données. Le traitement est légèrement dissymétrique. Les instructions `PROD_HI` et `PROD_LO` calculent respectivement le chiffre de poids fort et le chiffre de poids faible du produit des deux opérandes. Le reste du circuit calcule l'accumulation des produits partiels dans des jetons intermédiaires et le retrait des jetons définitifs de la boucle par l'instruction `POP_MULT`. Cette instruction utilise le champ exposant qui est modifié par l'opérateur `ADD_CARRY` pour détecter la fin de la boucle.

Nous présentons ici le code relatif à l'opération arithmétique de multiplication dans le simulateur logiciel. L'arbre est décrit à l'envers en partant de la racine. Il ne s'agit pas ici de fonctions mais d'opérateurs de construction d'un arbre. Les paramètres de chaque opérateur spécifient les noeuds auxquels sont envoyés le ou les résultats. Les symboles `right_man`, `left_man`, `out_man`, `right_exp`, `left_exp` et `out_exp` désignent les ports d'entrée et de sortie de l'opérateur. La première ligne signifie que les résultats de l'instruction `POP_MULT` sont envoyés sur la sortie `out_man` de l'opérateur

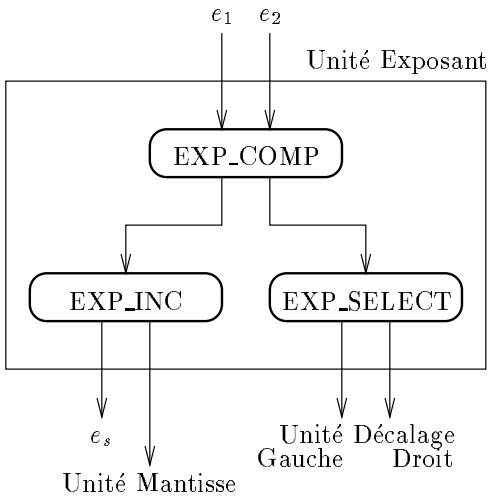


FIG. 2.27 - *Additionneur — Unité de Calcul sur les Exposants*

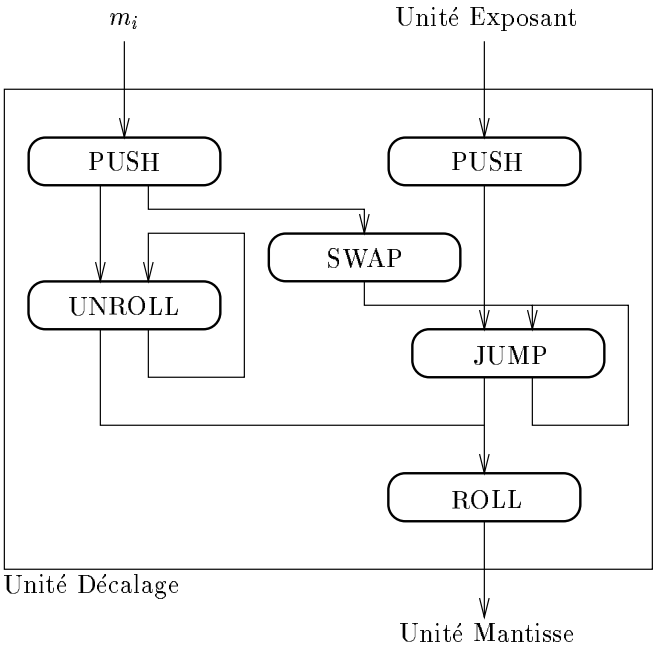


FIG. 2.28 - *Additionneur — Unité de Décalage des Mantisses*

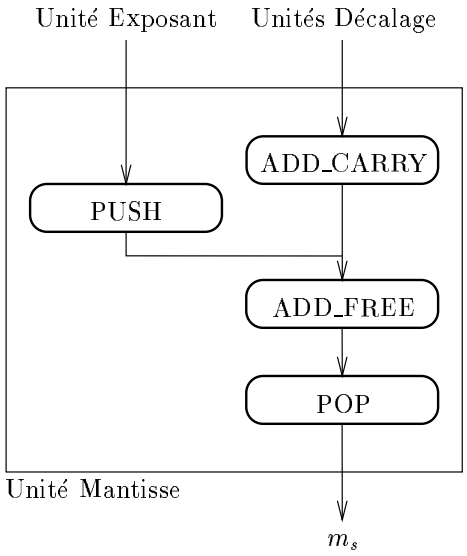


FIG. 2.29 - *Additionneur — Unité de Calcul sur les Mantisses*

et au port gauche de l'instruction `ADD_CARRY`. On voit ligne 2 que les jetons envoyés à l'instruction unaire `POP` sont générés par l'instruction `ADD_FREE`.

```
// Produits Partiels et Accumulations
pop       = POP_MULT  (out_man         , add_carry ->left);
add_free  = ADD_FREE  (pop      ->left                   );
add_carry = ADD_CARRY (add_free ->left , add_free ->right);
pp_hi     = PROD_HI   (add_carry->left , dup_l    ->left );
pp_lo     = PROD_LO   (add_carry->left , dup_r    ->left );

// Traitement Mantisse Droit
dup_r     = SWAP      (pp_hi    ->left , pp_lo    ->right);
zero_r    = ZERO      (add_free ->right                  );
roll_r    = ROLL      (zero_r   ->left                   );
push_r    = PUSH      (dup_r    ->left , roll_r   ->left );
right_man = push_r->left;

// Traitement Mantisse Gauche
dup_l     = DUP       (pp_lo    ->left , pp_hi    ->right);
zero_l    = ZERO      (add_free ->right                  );
push_l    = PUSH      (dup_l    ->left , zero_l   ->left );
left_man  = push_l->left;

// Unité Exposants
zexp      = PUSH      (add_carry->left                   );
incr      = EXP_INC   (out_exp         , zexp     ->left );
add_exp   = EXP_SUM   (incr     ->left                   );
```

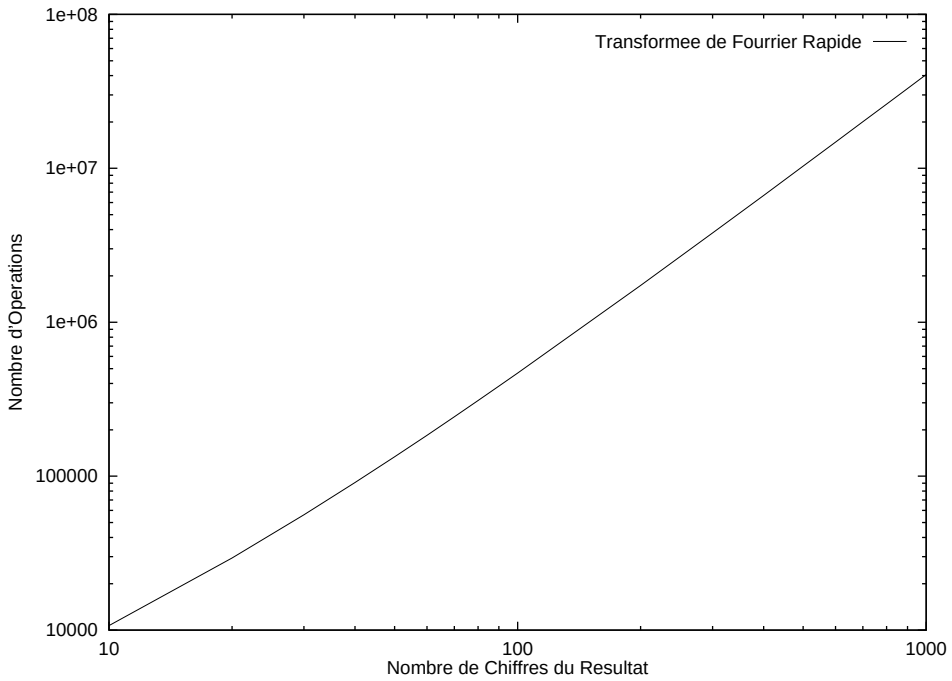


FIG. 2.30 - *Nombre d'Opérations pour Calculer une FFT sur un Calculateur Puce*

```
left_exp  = add_exp->left ;
right_exp = add_exp->right;
```

2.5.2 Simulation Logicielle

Afin de valider ces premiers travaux sur une architecture en-ligne à flot de donnée, j'ai réalisé un logiciel qui simule le fonctionnement d'un groupe d'unités Puce. Le logiciel nous permet de faire varier de nombreux paramètres d'une implantation de Puce. Le premier paramètre important est la base de numération. Nous avons opté dans ces travaux pour la base 10 afin de permettre une lecture aisée des résultats produits par le simulateur.

Le programme dont nous allons étudier la trace d'exécution calcule la transformée de Fourier discrète de huit nombres complexes flottants. Les données sont générées aléatoirement par le programme, bien que l'on puisse les générer par d'autres moyens. Le nombre de chiffres des opérandes est limité, par la taille du champ itération des jetons, à 2048. On peut néanmoins modifier cette grandeur aisément. Le nombre d'opérations élémentaires dans le calcul de la FFT, pour différentes tailles du résultat, est donné Figure 2.30.

Deux opérateurs sont utilisés par le programme. L'opérateur d'addition fait intervenir une boucle qui agit sur les chiffres des opérandes. À l'exception des opérations de normalisation, un opérateur d'addition génère un nombre linéaire d'instructions en fonction de la taille des données. L'opérateur de multiplication construit une double boucle imbriquée pour générer et accumuler l'ensemble des produits partiels. Dans le cas de la FFT, c'est cette opération qui est dominante. Pour cette raison, le nombre d'opérations élémentaires croît de manière quadratique avec la taille des données.

L'unité chargée d'apparier les jetons se comporte comme une mémoire associative. Nous avons vu pour le cas de la MDM qu'il n'est toutefois pas nécessaire de l'implanter effectivement avec une mémoire associative. On peut simuler plus ou moins efficacement une mémoire associative à

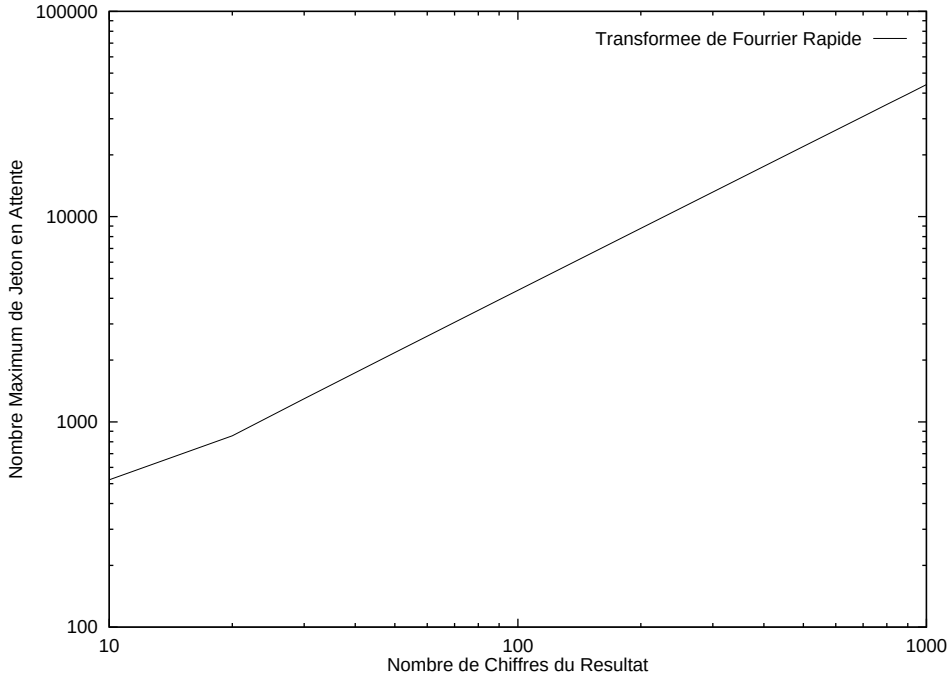


FIG. 2.31 - *Mémoire Associative utilisée pour le Calcul d'une FFT*

l'aide d'une mémoire conventionnelle et d'un bon mécanisme de hachage. Nous avons représenté Figure 2.31 l'évolution du nombre maximum de jetons en attente dans l'unité d'appariement en fonction du nombre de chiffres des résultats. On vérifie ici que la croissance de la fonction est linéaire.

Nous pouvons à l'intérieur du prototype logiciel de Puce faire varier le nombre d'unités fonctionnelles présentes dans le système. La Figure 2.32 montre à chaque instant le nombre d'unités fonctionnelles utilisées dans le calcul de 500 chiffres décimaux de la FFT en supposant que l'on dispose d'un nombre illimité d'unités dans le système. Les deux maximum d'activité se situent au tout début de l'exécution, quand les chiffres des opérandes arrivent et doivent être envoyés aux différentes unités (le segment apparaît mal sur le papier), et quand le dernier chiffre des opérandes arrive dans les multiplieurs et provoque les plus grands des produits entre un chiffre et un nombre. Après ce cap, les unités fonctionnelles ne calculent plus de produits partiels, et l'activité décroît au fur et à mesure que ceux-ci disparaissent dans l'accumulateur.

Afin de graduer l'échelle de temps des schémas, nous avons dû faire une hypothèse sur le fonctionnement d'une unité Puce. Nous avons adopté l'hypothèse très conservatrice d'une vitesse d'horloge de 10 MHz. Cette vitesse nous servira à évaluer le temps de calcul des programmes. Il est bien évidemment impossible de faire fonctionner 16984 unités puce ensemble : compte tenu de la complexité des unités fonctionnelles et de la vitesse des microprocesseurs actuels ainsi que de l'architecture des circuits reconfigurables que nous avons utilisés à ce jour, nous pouvons espérer intégrer un maximum de 10 ou 20 circuits fonctionnels dans un prototype matériel ou sur une implantation logicielle.

Nous avons présenté Figure 2.34 et Figure 2.35 des traces d'exécution pour le calcul de la FFT sur 500 chiffres décimaux. La seconde figure montre que même dans un cas où le nombre d'unités fonctionnelles est très élevé comparé à nos capacités actuelles, le pipeline au niveau du chiffre induit par le calcul en ligne est suffisant. Chaque graphe contient trois courbes selon la latence du réseau

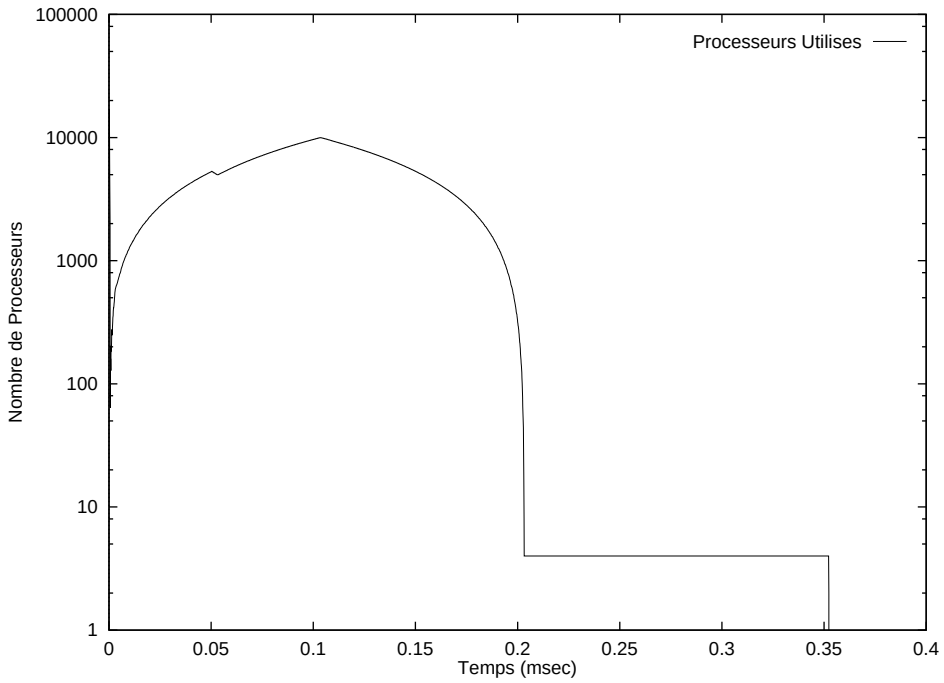


FIG. 2.32 - *Nombre d'Unité Puces Utilisées au Cours du Temps pour le Calcul d'une FFT*

et de l'unité d'appariement. Il est impossible d'envisager une mémoire associative de l'ordre de quelques centaines de Mmots avec une clef de recherche de 64 bits et un temps d'accès de 100 ns. De ce fait, on peut penser que dans des situations réelles, le stockage et la manipulation des jetons engendreront un temps d'attente. Dans le cas d'une implantation logicielle, cette durée représente le temps d'attente que l'on autorise pour garantir qu'aucun n'aléa n'aura lieu dans le pipeline d'exécution du microprocesseur.

La première courbe présente le comportement idéal d'une machine, avec 10 ou 100 unités fonctionnelles, capable d'apparier les jetons en temps nul. Pour tracer la seconde courbe, nous avons fait l'hypothèse plus réaliste d'un temps moyen de latence de 10 cycles entre le moment où le second jeton d'un paire est produit et le moment où il peut être consommé par une unité fonctionnelle. Il s'agit ici du temps moyen, les jetons à destination d'opérations unaires ne passent pas par l'unité d'appariement : on peut espérer d'une unité d'appariement hiérarchisée, un temps d'accès moyen de cet ordre.

On voit que le parallélisme est suffisant dans les deux cas et les deux courbes sont presques confondues. Les temps de fin de calcul (voir Figure 2.33) peuvent seuls permettre de distinguer les deux exécutions. La troisième courbe présente le comportement de la machine avec des hypothèses conservatrices, l'unité d'appariement dispose en moyenne de 5 ms pour constituer une paire si le premier jeton était déjà présent à l'arrivée du second. On voit que dans l'hypothèse peu probable où nous aurions intégré 100 unités fonctionnelles sur un calculateur Puce, la perte d'efficacité est de l'ordre de 50%. Pour le cas plus raisonnable ou l'on ne dispose que de 10 unités fonctionnelles, le parallélisme d'une seule FFT de 8 nombres est suffisant pour faire disparaître la latence.

Latence (cycles)	0	10	50
Nombre de processeurs			
10	103.4	104.9	111.0
100	010.4	011.9	019.7

FIG. 2.33 - *Temps de Calcul d'une FFT avec 500 Chiffres sur Puce (ms)*

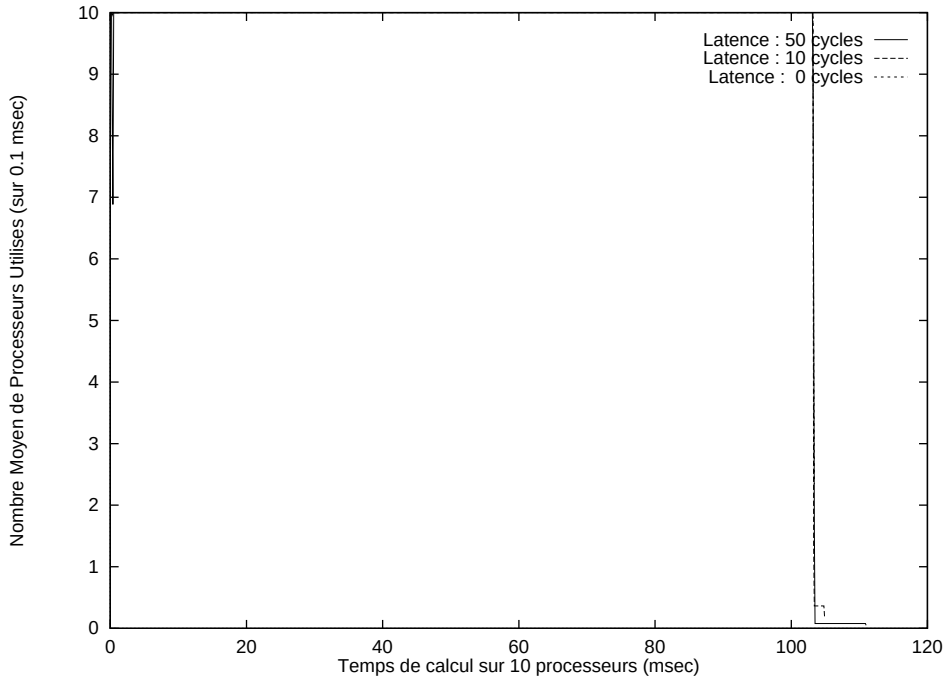


FIG. 2.34 - *Calcul d'une FFT sur 10 Processeurs Puces*

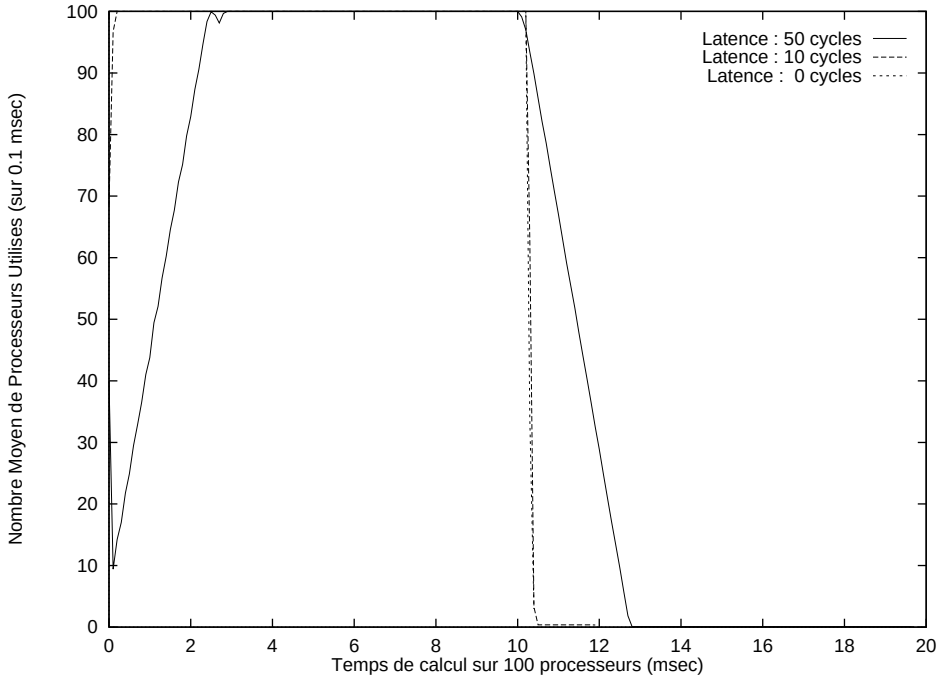


FIG. 2.35 - *Calcul d'une FFT sur 100 Processeurs Puces*

2.6 Quelle Arithmétique En-ligne pour Demain

Les méthodes, que nous avons mises en place, pour implanter les opérateurs en-ligne en notation BS sont maintenant bien maîtrisées. Pour chaque opération, nous sommes à même de concevoir un opérateur rapide, et optimisé en temps, à l'aide de la cellule Nacel. La conception du noyau Nacel en n'utilisant uniquement que les cellules des prédiffusés actifs nous a forcé à remettre en cause certains choix propres au développement de circuits VLSI. Mais avec les moyens qui sont actuellement en notre possession il n'est pas encore possible d'envisager une application en-ligne efficace sur prédiffusé actif.

Un opérateur général capable de calculer 72 chiffres du résultat à 20 MHz occupe la totalité d'un circuit. Cela nous limite énormément dans la pratique, et nous ne pouvons, dans les faits, qu'augmenter la taille de l'opérateur de multiplication. Il faut donc, pour obtenir une précision un peu plus élevée que l'arithmétique à virgule flottante, attendre au moins 3 ms. De nombreux travaux antérieurs dont [Pri 91] ont montré que l'on pouvait obtenir un résultat précis sur deux mots flottants en double précision en une dizaine d'opérations en langage machine. Ainsi, avec un ordinateur équipé d'un processeur DEC Alpha AXP cadencé à 100 MHz et capable de commencer une multiplication flottante à chaque cycle, on peut obtenir une précision supérieure à celle d'une cellule de calcul en-ligne en moins de 0.1 ms. Le pipeline au niveau du chiffre n'est utilisable qu'en présence de plusieurs opérateurs, mais comme pour obtenir une précision raisonnable, on doit consacrer tout un circuit à un opérateur, on ne peut espérer implanter plus de 16 opérateurs. En conservant des hypothèses très larges, cela nécessite moins de 2 ms sur une architecture conventionnelle.

L'opérateur d'addition occupe très peu de place car est il très simple. On pourrait envisager sur carte PeRLe un circuit en-ligne qui utiliserait principalement des additions et des multiplications. Avec une telle application, on pourrait espérer dépasser les performances d'un ordinateur conventionnel, même si les investissements en temps de développement et en coût matériel seraient

probablement trop importants en regard du gain de performance. Le problème est le même pour les circuits VLSI qui sont réalisés par d'autres groupes de recherche ; leurs prototypes calculent dans la pratique sur 16 ou 32 chiffres binaires signés. Pour obtenir de réels gains à l'aide de l'arithmétique en-ligne, il faut s'affranchir de la limite de taille des opérateurs pour pouvoir effectivement faire des calculs en grande précision.

Nous avons à la fin de ce chapitre décrit le fonctionnement d'une machine virtuelle capable d'évaluer simplement et de façon transparente pour l'utilisateur les expressions numériques à l'aide du calcul en-ligne. Chaque chiffre du résultat qui est produit par la machine est garanti. Pour obtenir plus de chiffres du résultat, il suffit de fournir plus de chiffres des données. On récupère alors des chiffres supplémentaires du résultat sans limite de taille autre que la mémoire disponible dans l'unité d'appariement des jetons. Une analyse de la mémoire permet de détecter quels sont les nombres qu'il faut spécifier avec plus de précision pour affiner le calcul de façon encore plus sélective. A tout moment, les chiffres produits sont garantis, et l'erreur est majorée par une unité en dernière position du dernier chiffre obtenu.

Nous avons implanté les deux opérateurs qui sont décrits ici dans un simulateur logiciel afin d'obtenir les courbes de mise à feu d'une addition et d'une multiplication selon le nombre d'éléments de calcul disponibles. Le calcul d'une addition demande un travail régulier dans le temps. Le nombre de jetons présents dans la mémoire associative est stable au cours du temps et le nombre de processeurs utilisés reste constant. Par opposition le calcul d'une multiplication génère un nombre croissant de jetons et le nombre de processeurs utilisés croît linéairement avec le nombre de chiffres des opérandes. Ces résultats conformes à notre attente montrent que le parallélisme induit par une addition est limité alors qu'une multiplication génère un parallélisme croissant. Pour exploiter pleinement Puce, il faut effectuer plusieurs additions ou quelques multiplications sur de grands nombres. Bien sûr, une expression usuelle comprend généralement plusieurs de ces opérateurs ainsi que des divisions, des extractions de racine carrée et des fonctions élémentaires. La division et la racine carrée seront obtenues dans le futur grâce au multiplieur et ces opérations présenteront alors le même comportement que le multiplieur. Nous pensons implanter les fonctions élémentaires grâce à l'algorithme CORDIC [Vol 59] dont l'implantation sur une arithmétique en-ligne en grande base a déjà été étudiée sur machine parallèle [Maz 93].

Nous espérons dans le futur pouvoir utiliser les travaux réalisés sur les machines à flot de données pour le développement d'une bibliothèque optimisée de Puce. Quand l'ensemble des opérations arithmétiques en-ligne auront été implantées sur le simulateur logiciel, nous pourrons envisager de réaliser un prototype logiciel de test. Il faudra définir le jeu d'instructions et l'implantation de chacune des unités Puces. L'évaluation d'expressions simples pourra être prise en charge sans problème dans le cadre d'une unité d'appariement stockée dans quelques lignes du cache du microprocesseur. Nous espérons implanter l'arithmétique en-ligne de manière transparente pour l'utilisateur en modifiant le compilateur d'un langage à affectation unique comme OSC pour le faire générer du code Puce en grande base sur les stations de travail et les machines vectorielles.

Conclusion

Ce manuscrit vous a présenté quelques possibilités d'évolution de l'arithmétique des ordinateurs, et les questions que je me suis posées en étudiant les ordinateurs modernes. Je n'ai pas tenté dans ce travail une étude exhaustive de toutes les évolutions possibles de l'arithmétique des ordinateurs, loin s'en faut. J'ai préféré aborder avec les chercheurs et les étudiants, avec qui il m'a été donné de travailler, des sujets relativements novateurs tels que le codage compact des intervalles et la description d'une Petite Unité de Calcul En-ligne. Si certains des sujets abordés dans ces travaux sont arrivés à maturité comme l'arrondi fidèle ou la construction d'opérateurs en-ligne sur un circuit reconfigurable, d'autres sujets sont pleins de promesses. Il est probable que l'on voit un jour une arithmétique d'intervalles transparente pour l'utilisateur implantée en machine. Même si cette implantation ne sera pas conforme aux propositions contenues dans cette thèse, les idées qui nous ont guidé dans la définition de notre arithmétique d'intervalle sont très attractives et seront sûrement reprise sous une forme plus avancées. Le développement des recherches autour de l'unité Puce semble très prometteur aussi bien pour la compréhension des mécanismes de l'arithmétique en-ligne que pour l'implantation d'une plateforme logicielle efficace est simple à utiliser de calcul en-ligne.

J'espère par ce travail, ainsi que par les publications auxquelles il a donné lieu, apporter ma pierre à l'élaboration des ordinateurs de demain. Pour le grand public, il s'agit de remettre en cause la course vers une précision quadruple si l'on ne se donne pas les moyens de contrôler au moins partiellement les erreurs engendrées. Nous avons ainsi fait la preuve que quelques opérations insérées astucieusement dans le cours des calculs pouvaient offrir un contrôle accru de la préision sans pénaliser l'utilisateur par un temps de calcul excessif.

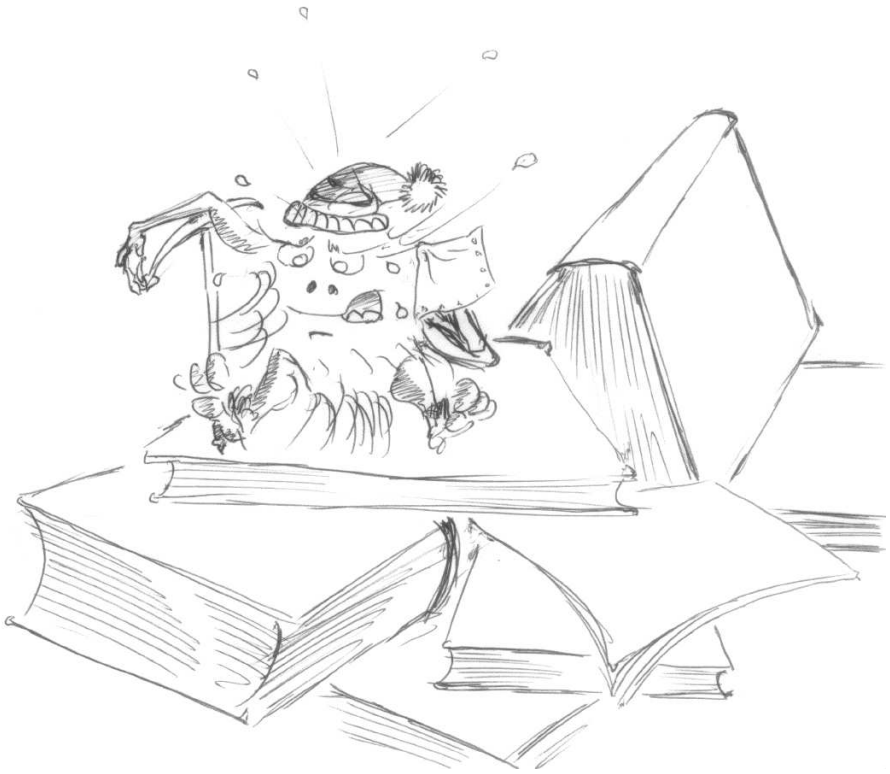
Pour les spécialistes, nous avons démontré avec le prototype logiciel de Puce et les réalisations sur l'accélérateur matériel PeRLe que l'arithmétique en-ligne peut être utilisée efficacement pour le calcul d'expressions complexes. La réalisation d'un circuit d'évaluation en-ligne d'une expression n'est toujours pas une tâche aisée même si l'on pourra espérer avec le développement de la techologie obtenir des opérateurs très rapides. Par contre, une fois implantés sur les microprocesseurs modernes, les mécanismes de Puce devraient être accessibles à tous sans que l'utilisateur ait besoin de connaître le fonctionnement interne en-ligne d'évaluation des expressions.

En continuant à travailler dans ces directions, nous pourrons dans un futur proche offrir à l'utilisateur des ordinateurs capables de calculer vite tout en calculant bien.

Le rapide élimine le lent, même si le rapide a tort.

William Kahan

Bibliographie



[Avi 61] ALGIRDAS AVIŽIENIS,
"Signed Digit Number Representations for Fast Parallel Arithmetic,"
IRE Transaction on Electronic Computers,
Vol EC-10, 1961.

[AC 86] ARVIND & DAVID E. CULLER,
"Dataflow Architectures,"
Annual Review in Computer Science,
Vol 1, 1986.

[AEGP 67] S.F. ANDERSON, J.G. EARLE, R.E. GOLDSCHMIDT & D.M. POWERS,
"The IBM System/360 Model 91: Floating Point Execution Unit,"
IBM Journal of Research and Development,
Vol 11, 1967.

[AGP 78] ARVIND, K.P. GOSTELOW & W. PLOUFFE,
"An Asynchronous Programming Language and Computing Machine,"
University of California at Irvine,
Technical Report CA 114a, 1978.

[AI 83] ARVIND & R.A. IANNUCCI,
"Instruction Set Definition for a Tagged-Token DataFlow Machine,"
Laboratory of Computer Science MIT,
Technical Report CSG 212-3, 1983.

[Bak 75] ALAN BAKER,
"Transcendental Number Theory,"
Cambridge University Press,
Cambridge, 1975.

[Ble. 87] H. BLEHER, S.M. RUMP, ULRICH KULISCH, M. METZGER, CH. ULLRICH & W. WALTER,
"Fortran-SC: A Study of a Fortran Extension for Engineering / Scientific Computation with Access to Acrith"
Computing,
Vol 39, 1987.

[Boh 77] GERD BOHLENDER,
"Floating Point Computation of Functions with Maximum Accuracy,"
IEEE Transactions on Computers,
Vol C26(7), 1977.

[Boh 90] —,
"What do we need Beyond IEEE Arithmetic?"
Computer Arithmetic and Selft Validating Numerical Methods,
Academic Press, 1990.

[BB 93] PATRICE BERTIN & PHILIPPE BOUCARD,
"DECPeRLe-1 Hardware Programmer's Manual,"
Digital Equipment Corporation, Paris Research Laboratory,
1993.

[BDKM 94] Jean-Claude Bajard, Jean Duprat, Sylvanus Kla & Jean-Michel Muller,
"Some Operators for On-Line Radix 2 Computation,"
Journal of Parallel and Distributed Computing,
Vol 22(2), 1994 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 92-42, 1992.

[BM 92] Jean-Claude Bajard & Jean-Michel Muller,
"A new VLSI Architecture for Fast On-line Evaluation of Power Series,"
International Conference on Signal Processing Application and Technology,
Boston — Massachussets, 1992.

[BM 94] Hans-Juergen Brand & Dietmar Mueller,
"Specification and Synthesis of Complex Arithmetic Operators for FPGAs,"
4th Field Programmable Logic Workshop,
Prague, 1994.

[BRV 89] Patrice Bertin, Didier Roncin & Jean Vuillemin,
"Introduction to Programmable Active Memories,"
Systolic Array Processors,
Prentice Hall, 1989 ;
Aussi disponible :
Digital Equipment, Paris Research Laboratory,
Rapport de Recherche 3, 1989.

[BRV 92] —,
"Programmable Active Memories: a Performance Assessment,"
1st ACM/SIGDA Workshop on Field Programmable Gate Arrays,
Berkeley — Californie, 1992 ;
Aussi disponible :
Digital Equipment, Paris Research Laboratory,
Rapport de Recherche 24, 1993.

[BWKM 91] Gerd Bohlender, W. Walter, Peter Kornerup & David W. Matula,
"Semantics for Exact Floating Point Operations,"
11th IEEE Symposium on Computer Arithmetic,
Grenoble, 1991.

[Che 88] Jean-Marie Chesnaux,
"Étude Théorique et Implémentation en ADA de la Méthode CESTAC,"
Université Paris VI,
Thèse, 1988.

[Che 95] Jean-Marie Chesnaux,
"L'Arithmétique Stochastique et le Logiciel CADNA,"
Université Paris VI,
Habilitation à Diriger des Recherches, 1995.

[Cod 73] WILLIAM J. CODY,
"Static and Dynamic Numerical Characteristics of Floating Point Arithmetic,"
IEEE Transactions on Computers,
Vol C22(6), 1973.

[Cod 87] —,
"Analysis of Proposals for the Floating Point Standard,"
IEEE Computer,
Vol 20(3), 1987.

[CD 88] B. CHOPARD & M. DROZ,
"Cellular Automata Approach to Non-Equilibrium Phase Transitions in a Surface Reaction Model: Static and Dynamic Properties,"
Journal of Physics,
Math. Gen. 21, 1988.

[CFB 93] FRANÇOISE CHATELIN, VALÉRIE FRAYSSÉ & THIERRY BRACONNIER,
"Qualitative Computation: Elements of a Theory for Finite Preision Computation,"
Workshop on Reliability of Computations,
Toulouse, 1993.

[CHS 80] D. COMTE, N. HIFDI & J. SYRE,
"The Data Driven LAU Multiprocessor System: Results and Perspectives,"
IFIP Congress 80,
Tokyo, 1980

[CR 78] C.Y. CHOW & J.E. ROBERTSON,
"Logical Design of a Redundant Binary Adder,"
4th IEEE Symposium on Computer Arithmetic,
1978.

[CV 92] JEAN-MARIE CHESNAUX & JEAN VIGNES,
"Les Fondements de l'Arithmétique Stochastique,"
Comptes Rendus de l'Académie des Sciences,
Série 1(315) , 1992.

[CW 80] WILLIAM J. CODY & WILLIAM WAITE,
"Software Manual for Elementary Functions,"
Prentice Hall,
Englewood Cliffs — New Jersey, 1980

[Dau 92] MARC DAUMAS,
"Basis for the Implementation of a Reliable Dot Product,"
Southern Methodist University,
Master Thesis, 1992.

[Dau 93] —,
"Un compilateur Sisal dérivé d'Osc sur Machine à Mémoire Logiquement Partagée,"
Journées des Jeunes Chercheurs en Systèmes à Mémoire Logiquement Partagée,
Toulouse, 1993.

[Den 74] Jack B. Dennis,
"First Version of a Data Flow Procedure Language,"
Colloque sur la Programmation,
Lecture Notes in Computer Science,
Vol 19, 1974.

[Den 80] —,
"Dataflow Supercomputers,"
IEEE Computer,
1980.

[DBL 80] Jack B. Dennis, G.A. Boughton & C.k-C. Leung
"Building Blocks for Data Flow Prototypes,"
7th Annual Symposium on Computer Architecture,
La Boule, 1980.

[DDT 94] Marc Daumas, Jérôme Durand & Louis-Pascal Tock,
"High Speed Implementation of Cellular Automaton,"
XIVth SC Chilean Conference,
Santiago du Chili, 1994.

[DE 93] Marc Daumas & Paraskevas Evripidou,
"Parallel Implementation of the Selection Problem using Sisal,"
IFIP Working Group 10.3 - Working Conference on Concurrent Systems,
Orlando — Floride, 1993.

[DM 91] Jean Duprat & Jean-Muller,
"Écrire les Nombres Autrement pour Calculer plus Vite,"
Techniques et Science Informatiques,
1991.

[DM 93a] Marc Daumas & David W. Matula,
"Design of a Fast Validated Dot Product Operation,"
11th IEEE Symposium on Computer Arithmetic,
Windsor — Ontario, 1993.

[DM 93b] —,
"Rounding of Floating Point Intervals,"
IMACS/GAMM International Symposium SCAN-93,
Vienne, 1993 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 93-06, 1993.

[DMM 94] Marc Daumas, Christophe Mazenc & Jean-Michel Muller,
"User Transparent Interval Arithmetic,"
IMACS/GAMM International Symposium SCAN-93,
Vienne, 1993 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 94-02, 1994.

[DMMM 95] Marc Daumas, Christophe Mazenc, Xavier Merrheim & Jean-Michel Muller,
"Modular Range Reduction: a New Algorithm for Fast and Accurate Computation of the Elementary Funcitons,"
Journal of universal Computer Science,
Vol 1(3), 1995 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 95-03, 1995 ;
Sujet étendu de :
"Fast and Accurate Range Reduction for Computation of Elementary Functions,"
14th IMACS World Congress,
Atlanta — Georgie, 1994.

[DMV 94] Marc Daumas, Jean-Michel Muller & Jean Vuillemin,
"Implementing On-Line Arithmetic on Pam,"
4th Field Programmable Logic Workshop,
Prague, 1994 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 94-25, 1994.

[Erc 77] Miloš D. Ercegovac,
"A General Hardware Oriented Method for Evaluation of Functions and Computations in a Digital Computer,"
IEEE Transactions on Computers,
Vol C26(7), 1977.

[Erc 84] —,
"On Line Arithmetic: an Overview,"
SPIE, Real Time Signal Processing VII,
1984.

[EMT 95] Miloš D. Ercegovac, Jean-Michel Muller & Arnaud Tisserand,
"FPGA Implementation of Polynomial Evaluation Algorithm,"
Spie,
Philadelphie — Pennsylvanie, 1995.

[FB 91] Warren E. Ferguson Jr. & Tom Brightman,
"Accurate and Monotone Approximations of some Transcendental Functions,"
10th IEEE Symposium on Computer Arithmetic,
Grenoble, 1991.

[FCO 90] John Feo, David Cann and R. Oldehoef,
"A Report on the Sisal Language Project,"
Journal of Parallel and Distributed Computing,
Vol. 10(4), 1990

[FE 93] Wiiliam G. Farquhar & Paraskevas Evripidou,
"DART: A Data-Driven Processor Architecture for Real-Time Computing,"

IFIP Working Group 10.3 - Working Conference on Concurrent Systems,
Orlando — Floride, 1993.

[FG 76] Alan Feldstein & Richard Goodman,
"Convergence Estimates for the Distribution of Trailing Digits,"
Journal of the Association for Computing Machinery,
Vol 23(2), 1976.

[FM 90] Philippe François & Jean-Michel Muller,
"Faut-il faire Confiance aux Ordinateurs?"
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 90-03, 1990.

[Gau. 85] Jean-Luc Gaudiot, Rex W. Vedder, George K. Tucker, Dennis Finn & Michael L. Campbell,
"A Distributed VLSI Architecture for Efficient Signal and Data Processing,"
IEEE Transactions on Computers,
Vol C34(12), 1985.

[Gol 90] David Goldberg,
"Floating Point Arithmetic,"
Computer Architecture: a Quantitative Approach,
1990.

[Gol 91] —,
"What Every Computer Scientist Should Know About Floating-Point Arithmetic,"
ACM Computing Surveys,
Vol 23(1), 1991.

[Gos 71] J.B. Gosling,
"Design of Large High Speed Floating Point Arithmetic Units,"
Institution of Electrical Engineers Proceeding,
Vol 118, 1971.

[Gwe 95] Linley Gwennap,
"Intel's P6 Uses Decoupled Superscalar Design,"
Microprocessor Report,
1995.

[GHM 89] Alain Guyot, Yvan Herreros & Jean-Michel Muller,
"Janus, an On-Line Multiplier/Divider for Manipulating Large Numbers,"
9th IEEE Symposium on Computer Arithmetic,
Santa Monica — Californie, 1993.

[GKW 85] J.R. Gurd, C. Kirkham & I. Watson,
"The Manchester Prototype Dataflow Computer,"
Communications of the Association for Computing Machinery,
1985.

[IEEE 85] American National Standards Institute & Institute of Electrical and Electronics Engineers,

"IEEE Standard for Binary Floating Point Arithmetic,"
ANSI/IEEE Standard,
Std 754-1985, New York, 1985.

[IEEE 87] American National Standards Institute & Institue of Electrical and Electronics Engineers,
"IEEE Standard for Radix Independent Floating Point Arithmetic,"
ANSI/IEEE Standard,
Std 854-1987, New York, 1987.

[Kla 93] Sylvanus Kla,
"Calcul Parallèle en Ligne des Fonctions Arithmétiques,"
Laboratoire de l'Informatique du Parallélisme,
Thèse 93-01, 1993.

[KM 81] Ulrich Kulisch & Willard L. Miranker,
"Computer Arithmetic in Theory and Practice,"
Academic Press,
1981.

[Lyn 95] Thomas Lynch,
"High Radix Online Arithmetic for Credible and Accurate General Purpose Computing,"
Real Number and Computers,
Saint-Étienne, 1995.

[LE 93] Marianne E. Louis & Miloš Ercegovac,
"On Digit Recurrence Division Implementation for Field Programmable Gate Arrays,"
11th IEEE Symposium on Computer Arithmetic,
Windsor — Ontario, 1993.

[LS 93] Thomas Lynch & Earl E. Swartzlander,
"A Formalization for Computer Arithmetic,"
Computer Arithmetic and Enclosure Methods,
1993.

[LV 74] Michel La Porte & Jean Vignes,
"Évaluation Statistique des Erreurs dans l'Arithmétique des Ordinateurs, Application au Contrôle des Résultats d'Algorithmes Numériques,"
Numerische Mathematik,
Vol 23, 1974.

[Maz 93] Christophe Mazenc,
"Systèmes de Représentation des Nombres et Arithmétique sur Machines Parallèles,"
Laboratoire de l'Informatique du Parallélisme,
Thèse 93-10, 1993.

[MGr 83] McGraw *et al.*,
"SISAL — Streams and Iterations in a Single Assignment Language,"

"Language Reference Manual",
Lawrence Livermore National Laboratory,
1983.

[Mul 89] Jean-Michel Muller,
"Arithmétique des Ordinateurs,"
Masson,
1989.

[Mul 94] —,
"Some Characterizations of Functions Computable in on Line Arithmetic,"
IEEE Transactions on Computers,
Vol 43(6), 1994 ;
Aussi disponible :
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 91-15, 1991.

[MK 85] David W. Matula & Peter Kornerup,
"Finite Precision Rational Arithmetic: Slash Number Systems,"
IEEE Transactions on Computers,
Vol C34(1), 1985.

[MM 73] John D. Marasa & David W. Matula,
"A Simulative Study of Correlated Error Propagation in Various Finite Precision Arithmetic,"
IEEE Transactions on Computers,
Vol C22(6), 1973.

[MMP 95] Anne Mignotte, Jean-Michel Muller & Olivier Peyran,
"Mixed Arithmetic and Operations,"
Laboratoire de l'Informatique du Parallélisme,
Rapport de Recherche 95-17, 1995.

[MRM 93] Javier Moran, Ignacio Rios & Juan Meneses,
"Signed Digit Arithmetic on FPGAs,"
3th Field Programmable Logic Workshop,
1993.

[Pic 72] Michèle Pichat,
"Correction d'une Somme en Arithmétique à Virgule Flottante,"
Numerische Mathematik,
Vol 19, 1972.

[Pic 76] —,
"Contributions à l'Étude des Erreurs d'Arrondi en Arithmétique à Virgule Flottante,"
Université Scientifique et Médicale de Grenoble,
Thèse, 1976.

[Pri 91] Douglas M. Priest,
"Algorithms for Arbitrary Precision Floating Point Arithmetic,"

10th IEEE Symposium on Computer Arithmetic,
Grenoble, 1991.

[PV 93] Michèle Pichat & Jean Vignes,
"Ingénierie du Contrôle de la Précision des Calculs sur Ordinateur,"
Éditions Technip,
Paris, 1993.

[Sha 92] Mark Shand,
"Measuring System Performance with Reprogrammable Hardware,"
Digital Equipment, Paris Research Laboratory,
Rapport de Recherche 19, 1992.

[Ska 95] Ali Skaf,
"Concpetion de Processeurs Arithmétiques Redondants En-ligne : Algorithmes, Architectures et Implantations VLSI,"
Institut National Polytechnique de Grenoble,
Thèse, 1995.

[Sla 95] Michael Slater,
"MicroUnity Lifts Veil on MediaProcessor,"
Microprocessor Report,
1995.

[Stu 80] F. Stummel,
"Rounding Error Analysis of Elementary Numerical Algorithms,"
Computing,
Suppl. 2, 1980.

[SM 95] André Seznec & Yann Mével,
"Évolution des Gammes de Processeurs MIPS, DEC Alpha, PowerPC, SPARC et xxx86,"
Institut de Recherche en Informatique et Systèmes Aléatoires,
Publication interne 957, 1995.

[SS 93] Michael Schulte & Earl E. Swartzlander,
"Exact Rounding of Certain Elementary Functions,"
11th IEEE Symposium on Computer Arithmetic,
Windsor — Ontario, 1993.

[Tou 92] Hervé Touati,
"Perle1DC: a C++ Library for the Simulation and the Generation of DECPeRLe-1 Designs,"
Digital Equipment, Paris Research Laboratory,
Notes Techniques 4, 1992.

[TE 77] Kishor S. Trivedi & Miloš D. Ercegovac,
"On Line Algorithms for Division and Multiplication,"
IEEE Transactions on Computers,
Vol C26(7), 1977.

[THH 80] T Temma, S. Hasegawa & S. Hanaki,
"DataFlow Processor for Image Processing,"
11th Intenational Symposium on Mini and Microcomputers,
Monterey — Californie, 1980.

[USA 92] United States General Accounting Office,
"Patriot Missile Defense — Software Problem led to System Failure at Dahran, Saudia Arabia,"
Information Management and Technology Division,
Washington — District of Columbia, 1992.

[Vig 87] Jean Vignes,
"Zéro Mathématique et Zéro Informatique,"
Comptes Rendus de l'Académie des Sciences,
1987.

[Vol 59] J. Volder,
"The CORDIC Trigonometric Computing Technique,"
IRE Transactions on Electronic Computing,
Vol EC-8, 1959.

[Vui 92] Jean Vuillemin,
"Contribution à la Résolution Numérique des Équations de Laplace et de la Chaleur,"
À paraître :
Mathematical Modelling and Numerical Analysis;
Aussi disponible :
Digital Equipment, Paris Research Laboratory,
Rapport de Recherche 16, 1992.

[WF 82] Shlomo Waser & Michael J. Flynn,
"Introduction to Arithmetic for Digital Systems Designers,"
Holt, Rinehart and Winston,
Orlando — Floride, 1982.

[Xil 94] Xilinx inc.,
"The Programmable Gate Array Data Book,"
Xilinx Product Briefs,
San Jose — Californie, 1994.

[YSHK 85] Toshitsugu Yuba, Toshio Shimada, Kei Hiraki & Hiroshi Kashiwagi,
"Sigma-1: A DataFlow Computer for Scientific Computations,"
Computer Physics Communications,
Vol 37, 1985.

Table des figures

Table des matières